A TOOL FOR AURALIZED DEBUGGING

By

YAN CHEN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of YAN CHEN find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

A TOOL FOR AURALIZED DEBUGGING

Abstract

by YAN CHEN, M.S.
Washington State University
AUGUST 2005

Chair: Kelly Fitz

While graphical representations have been incorporated into program debugging tools, auditory displays have rarely been tried. Nevertheless, sound is inherently a temporal medium rather than a spatial one, so it is logical to explore the possibilities of representing the temporal evolution of program state in patterns of sound.

In this thesis, I consider the use of enhanced debugging environments supporting auditory representations of program state to improve the debugging performance of students in introductory computer programming classes. A pilot study of novice programmer debugging behavior and common logic errors is discussed. I describe a prototype auralizing debugger, and propose an experiment to see whether the sort of auralizations that can be generated with the tool are understandable to novice programmers.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

This thesis is dedicated to my baby boy Johnny Chen Hang,

my wife Xiaoju Hang, my parents Baixin Chen and Dehua Chen.

CHAPTER ONE

INTRODUCTION

1.1 Research Objectives

It has been shown, and my experience suggests, that program state is one of the most difficult aspects of program execution for novice programmers to understand [51, 52]. Graphical representations of program state have been proposed as an alternative to traditional textual interfaces [38]. The evolution of program state is a *temporal* phenomenon. Sound, like animation in graphical representation, is also a temporal medium and it offers an alternative output channel to explore. The mechanisms of auditory perception tend to focus our attention on the dynamic aspects of an auditory scene, and ignore the aspects that are unchanging.

It has been suggested that the properties of sound and listening make auditory representations ideally suited to improving program comprehension [24, 2, 14]. Auditory display, though rarely used in program development and debugging applications, may, therefore, be a sensible alternative to graphical representations of evolving program state.

To determine whether and how auditory representations of program state and execution can help students form a more complete mental model of the evolving state of a program, it is necessary for me to construct a new software tool to enable the auralization of program state and execution. So I have developed a prototype software tool for non-invasive auralization of program state and execution. Ultimately, I hope to develop sonified debugging tools that can improve the debugging performance of students in introductory computer programming classes.

While students in introductory computer programming classes may understand that a program does not work, and may even know which general part of the program is at fault, they often have insufficient understanding of how the program executes. They also often fail to develop a good mental model of the evolving state of a program, especially when advanced data structures are

involved.

This work investigates whether auditory representation of program state and execution can help students form a more complete mental model of the evolving state of a program.

1.2 Contributions

The designs of new debugging tools have generally not taken advantage of what has been discovered in the areas of empirical studies of programmers (ESP). Brad A. Myers did a thorough investigation of the ESP literature which revealed many results can be used to guide the design of new programming systems. However, he found that there are still many significant gaps in the knowledge about how people reason about programs and programming. It is in this context that Myers developed the Natural Programming environment design process [44], which treats usability as a first-class objective by following these steps:

- Identify the target and the domain, that is, the group of people who will be using the system and the kinds of problems they will be working on.
- Understand the target, by studying the actual techniques and the natural thought patterns used by novices when they try to solve problems. This includes prior work in the psychology of programming and empirical studies. When issues or questions arise that are not answered by the prior work, conduct new empirical studies to examine them.
- Use the knowledge to develop a new tool for debugging.
- Evaluate the system to measure its success, and to understand any new problems the users have. Redesign the system based on this evaluation, and then reevaluate it, following the standard HCI principle of iterative design.

My goal is to make the enhanced sound debugger more natural, reflecting the way novices think about their debugging jobs. To do this, I will follow the steps of the natural programming design process. First, I did an investigation of ESP literature about novice programmers on program

comprehension and debugging behavior. Then due to some issues that were not answered by the prior work, I ran a pilot study on novice programmers' debugging and programming activities, in which student debugging and programming activities were monitored and recorded. This data will be used to identify logic errors that students commonly introduce into their programs and detect common unproductive debugging practices. Then the findings from the pilot study guided the development and deployment of a new set of audio debugging tools to help students identify common programming errors more quickly. This will also discourage unproductive debugging activities. Because there is not enough time to perform the last step of natural programming environment design process, I will provide an example experiment for determining the comprehensibility issues of auralizations based on different schemes, on the different logic errors. This experiment is important in audio design spaces. Before we can perform the experimentation on the auralizing debugger, we first need to establish an semantic music framework to describe the best ways to map between program state and sound. There will be some experimentations like the one I will describe in chapter 6 on the comprehensibility issues of auralization. After building the music framework, developer need to improve the auralizing debugger from the prototype to a nearly-released version, and in this step there will be some experimentations on the useability issues. After these two steps, we can perform the last step of the natural programming design process. My current prototype auralizing debugger make it possible to perform the steps of building the music framework and improving to the nearly-released version.

1.3 Overview of Thesis

Chapter 1 consists of this introduction.

Chapter 2 surveys prior research on behavior of novice programmers and the limitations of prior research.

Chapter 3 describes why the pilot study on novice programmers' behavior and common errors was conducted, and how it was carried out. It also summarizes the results and findings.

Chapter 4 surveys the use of non-textual techniques, graphical display, and auditory display in the field of debugging.

Chapter 5 describes the motivation for and design of my audio-enhanced debugger system, which is a prototype for obtaining audio representation of the program state.

Chapter 6 proposes an experiment to discover whether the sound representation generated with my proposed tool would be understandable to novice programmers. Several auralization examples are provided.

Chapter 7 summarizes the work described in this thesis and suggests future research.

CHAPTER TWO

EMPIRICAL STUDIES OF NOVICE PROGRAM COMPREHENSION AND DEBUGGING BEHAVIOR

2.1 Introduction

The use of graphical techniques of program state [57] has been well developed and some very sophisticated visualization systems now exist. While much effort is being put into the development of graphical techniques for enhancing program understanding, the audio channel remains relatively little researched for its potential in assisting in the programming and debugging process. It is only recently that this medium is being explored by researchers and practitioners in human-computer interaction applications.

My goal is to make the enhanced sound debugger more natural, reflecting the way novices think about their debugging jobs. To do this, I will follow the steps of the natural programming environment design process. In this chapter, I will follow first two steps of the natural programming environment design process and review knowledge gained from empirical studies on novice programmer's program comprehension and debugging behavior, both of which are central to debugging.

2.2 Program Comprehension

Program comprehension is the process of understanding programs that are often written by other programmers. Program comprehension is critical to my research, because program debugging relies on it. Programmers need to comprehend faulty programs before they can identify bugs. In order to comprehend a program, programmers need to navigate the program code and extract different kinds of program information such as control flow and data flow information. The programmer assimilates and organizes the information to form an abstraction of the program that researchers refer to as the mental representation, or mental model, of the program.

Let's first review the influential models of text comprehension from which Pennington's well known model on mental representation was derived.

- The lowest layer in the mental model is the surface form representation, which is the reader's verbatim memory of the text.

- The middle layer in the mental model is the textbase representaion, which is more abstract than the surface form representation and contains microstructure and macrostructure knowledge based on propositions and relationships in the text.

- The top layer in the mental model is the more abstract situation model which contains not only the information in the text, but also the inferences about the text. The richness of the inference depends on the reader's own knowledge about the domain, and also the effort made by the reader.

Next I will discuss Pennington's model of program comprehension.

2.2.1 Pennington's Mental Representation

Pennington's mental representation [45, 46] model identifies five basic categories of program information making up the programmer's mental representation.

- Elementary Operations. Elementary operations are the basic text units usually represented by one or a few lines of code. For example, setting a variable's value would be an elementary operation. Operation information represents the information that is explicitly available in text, so it is a middle level of abstraction.

- Control Flow. Control flow involves the execution order of the source code. It represents the logical order of the program rather than the physical order. The default order is sequential, but it might be changed by functions calls etc. Control flow information represents the information that is explicitly available in text, so it is also a middle level abstraction.

- Data Flow. Data flow is concerned with how variables are manipulated as a program executes. This transformation of variables changes data from the initial input state to the final output state during execution, so that it is closely linked to the program goals. This information can not be obtained based solely on the information in the text, so it is a top level of abstraction.

- Function. Function information concerns the goals of a program. Understanding of the program's function may depend on the reader's prior knowledge of the program domain as well as personal effort. This information can involve multiple program units, and unlike element operation or control flow, which can be abtained by reading just a single line of code or a few lines of code, function information is not low abstraction but instead a top level of abstraction.

- State. State information is defined as the connection between execution of an action and the state of all aspects of the program that are necessarily true at a certain point in time.

Beyond the surface form representation of the program text, based on the level of abstraction, Pennington put the first four categories into two models (A model is an abstraction of some kind of information of program). The program model is analogous to the textbase in influential models of text comprehension, while the domain model is analogous to the situation model.

The program model is made up of elementary operations and control flow information which represent procedural aspects of a problem. The domain model consists of data flow and function information. It is based not just on information in the code, but also on the programmer's domain knowledge. and the effort made by the programmer to draw inferences from the code.

2.2.2 Programming Paradigms and Program Comprehension

Programming success can be enhanced if the programming language is closer to the novice's concept of system operation [44]. However, novices are usually advised to start with an imperative

language, regardless of suitability.

Corritore [15] researched program comprehension by novice imperative programmers. Results indicated that novice programmers comprehending an imperative style program form a strong program model. This was shown in the experiments by the low error rate in the imperative style on both operations and control flow questions, which together make up the program model. The state information, which Pennington did not assign to either the program or domain model, had a high error rate in experiments. This implies that state information is difficult to extract from programs and not readily available in programmers' mental representations of imperative programs.

Research has found that with different programming paradigms, novices have different program comprehension [43]. Object-oriented programming is a growing paradigm in computer science and becoming increasingly popular, but it appears that studies of object-oriented programming have concentrated primarily on program design as well as reuse and maintenance of experts. people know relatively little about the problems of learning programming in object-oriented paradigm, as compared to the substantial body or work on how students learn procedural and declarative forms of programming. Vennila Ramalingam [52] ran an empirical study of novice program comprehension in the imperative and object-oriented styles, finding a sharp contrast between mental representations of imperative and object-oriented programs [52]. The mental representation of imperative programs focused on program-level information, confirming the result of Corritore's experiment. On the other hand, mental representations of object-oriented programs focused on a strong domain model. The error rate was low on data flow and function questions, while state questions had high error rates, similar to the imperative program result of past research.

2.3 Debugging Behavior

The study of debugging behaviors is very important, because educators can use this research to enhance program development, debugging tools, and teaching methods in order to improve their students' debugging skills. As a result, students can gain confidence in their debugging abilities

and spend less time finishing their programming assignments.

Researchers observed that novices tend to have the following behavior:

- Novice programmers state more false hypotheses about bugs, make extensive modifications, and add additional bugs in the course of looking for current bugs. This makes it even more difficult for novices to isolate actual bugs [29, 47, 26].
- Novice programmers work with print-outs of the program and of output from test runs, often not using computers at all for this step [35, 36].
- Novice programmers repeat the same types of defects frequently throughout a program [12].
- Novice programmers debug programs haphazardly instead of following a plan [12].
- Novice programmers give up easily and depend on others for assistance [12].
- Before generating and testing an initial hypothesis, novices distribute their activities in the same way as experts, studying the program intensively before making changes. This step is not qualitatively different than expert behavior, but novices take longer [29].
- Novice programmers tend to use a sequential method of program navigation [34, 40] and view the program in a bottom-up fashion.
- Initially, novices seem to attempt a comprehension strategy, later shifting to an isolation approach in which they immediately attempt to identify candidate bug locations by searching the output for clues, recalling similar bugs, and testing program states [47].
- Novice programmers correct semantic errors first and then logic errors [47].
- Novice programmers correct and verify single errors [47].
- Novice programmers lack debugging skills [12]. For example, they might not know to trace program state and execution by setting a breakpoint.

2.4 Conclusion

Research on program comprehension and debugging behavior among novice programmers has shown that state information in mental representations is difficult to extract from the programs themselves. This state information is also not readily available in programmers' mental representations of object-oriented programs. Usually, novice programmers develop an understanding of local program behavior but fail to develop a mental model of the behavior of the program as a whole. My research seeks to determine whether and how auditory representations of program state can help students form a more complete understanding of the behavior of their programs, and consequently improve their programming and debugging performance.

CHAPTER THREE

PILOT STUDY ON NOVICE PROGRAMMERS' COMMON ERRORS AND DEBUGGING BEHAVIOR

3.1 Introduction

In introductory programming classes, I find that while students may understand that a program does not work, and may even know which general part of the program is at fault, they still have insufficient understanding of how the program executes and fail to develop a good mental model of the evolving state of a program. This is especially common when advanced data structures are involved.

I conducted a pilot study on novice programmers' debugging and programming activities in Fall, 2004, in which student debugging and programming activities were monitored and recorded. This data will be used to identify logic errors that students commonly introduce into their programs and detect common unproductive debugging practices In this chapter, I will describe the motivation for this study and how it was carried out.

3.2 Motivation of the Pilot Study

Research on novice debugging practice has been conducted using imperative programming languages such as Pascal [29, 35, 36, 47, 27] and Cobol [26], functional programming languages such as Lisp [13]and Logo [11, 62], and assembly language [12]. Research on programming in the Object-oriented (OO) style began to appear around 1995. However, studies of object-oriented programming have concentrated primarily on program design, and secondarily on reuse and maintenance of experts. Object-oriented languages (such as C++ and Java) are increasingly popular in computer science curricula, and these languages present a different set of conceptual barriers to students [52].

There are several reasons why prior studies don't generate the data I want.

- Those studies did not examine the role of debugging in novice program development.

  Studies of debugging behavior of both skilled and novice programmers (for example [29, 65, 67]) typically involve programs written by the author of the study and provided to the participant programmer, not programs written by the participants themselves. This research model provides a reasonable context for studying program comprehension, and it simulates a significant part of the professional activity of computer programming, since software maintenance groups are often required to debug unfamiliar programs. Reuse activities in modern software development also forces programmers to read, use, and debug code that they did not write.

  These studies cannot, however, explore the important role that debugging plays in the problem solving and development activities of novice programmers. Students of computer programming are often required to write programs from scratch, and debugging is part of that process. Students spend much of their time debugging their programs as they write them.

- Those studies used simple proposed problems, small programs with less object-oriented features.

  In prior experiments, the proposed problems were usually quite simple and the programs were very small. Certain important defining features of the object-oriented style were also not required, which include templates, overloading, object construction and copy control, inheritance and polymorphism.

- Those studies used inaproperate methods to collect their data.

  Moreover, programmer behavior is often an enigma. Researchers do not often know what programmers are thinking about, and tend to guess based on the data they do have. In many studies, researchers do not collect related elaborate information which can effect results. These factors often bias results drawn from statistical analyses.

In order to solve these problems, I have conducted a pilot study to help me understand how students develop and debug their programs. I collected time-stamped activity logs that were generated automatically by a specially-instrumented integrated development and debugging environment as students worked on their programming assignments. The assignments featured non-trivial object-oriented programs, using key language features like C++ templates, copy control, and polymorphism. The logs allowed me to trace the evolution of the students' program code, to measure the amount of time students spent running and debugging their programs, and to evaluate their use of breakpoints and other debugger features, without interrupting normal work flow.

The data gathered in this study will improve my understanding of how students integrate debugging into their programming activities, and allow me to re-evaluate the role of debugging in the undergraduate computer science curriculum, where it is often ignored [12]. Armed with this new information, I can focus both my teaching efforts and my development efforts on common programming problems and bad debugging habits.

3.3 Classifying Logic Errors

My research is focused on the logic errors novice programmers commonly introduce to their programs. These are the errors that solely depend on programmers' logic. In the literature, programming errors have been categorized according to the root cause of the error [37], the frequency of the error [21], or the amount of time [28] taken to find and remove the error. But such factors as the type of programming assignment, program size, program complexity and programmer's knowledge can cause the same error to be categorized differently in different contexts. For example, a memory-overwrite error might be found quickly in the context of array manipulation, but might be very difficult to find in the context of binary tree manipulation. If the programmer's tests are too weak, the error might go unnoticed.

In my research, I use the terms of *error*, *fault*, and *failure* from the IEEE standard to classify errors. The terms *error*, *fault*, and *failure* are defined in the IEEE standard 610.12-1990. *Errors* are

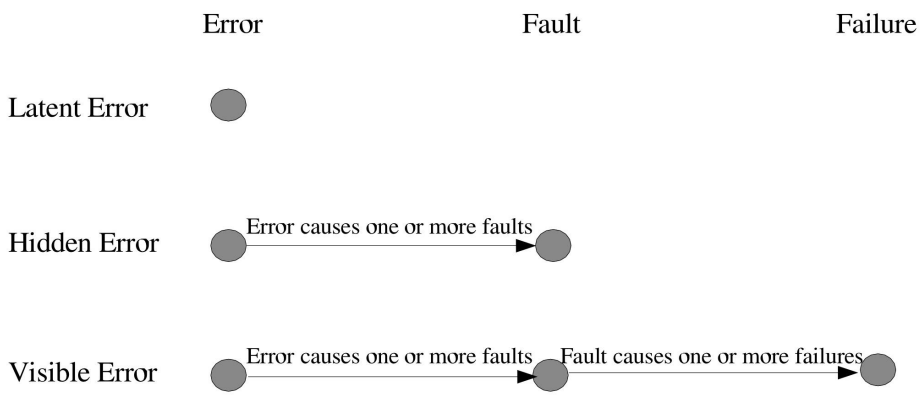

Figure 3.1: The three categories of logic errors.

program fragments that are inconsistent with program specifications. An error can result in a *fault*, which is a runtime state of the program that either is or appears to be incorrect. A fault implies the existence of at least one error. A *failure* is the departure of program behavior from the program requirements. A failure implies the existence of at least one fault.

Errors often cause faults, but faults may not lead to failures if the faulty state does not affect the visible behavior of the program. I call such errors *hidden errors* to distinguish them from *visible errors*, whose effects are visible in the failed program output. In some cases, an error may not even produce a fault, if the code that is in error is not executed (for example, if the test coverage is insufficient). I call this a *latent error*. The relationship between these three categories of errors, used in my pilot study, is shown in Figure 3.1.

3.4 Research Questions

This study analyzes the logic flaws that students commonly introduce into their programs and the techniques they use to identify logic errors for the purpose of evaluation with respect to the efficiency of debugging strategies and conventional debugging tool on common logic flaws.

3.5 Research Design

3.5.1 Type of Study

This study is an observational and on-line study, and it is a informal, non-rigorous, and uncontrolled investigation. The purpose of this study is to direct my future research. There is no hypothesis or theory motivating the pilot study.

3.5.2 Validity Threats

I do not worry much about threats to the validity of the study. Because it was an informal pilot study, the representativeness of certain students and assignments was not a concern.

To avoid affecting students' behavior during programming, I did not use think-aloud protocol. Therefore, it was sometimes hard to determine whether students involved in the pilot were debugging or testing programs.

3.6 Participants

Thirty-eight participants took part in my pilot study. All were volunteers. The participants were students in the School of Electrical Engineering and Computer Science at Washington State University, enrolled in Advanced Data Structures and C++ in Fall Semester, 2004. All students attended the same lectures twice each week, as well as one of four recitation sections once each week. Recitation sections were led by one of three teaching assistants under the coordination of the professor, who ensured that the sections covered the same material at approximately the same pace. All students used the same textbooks, and performed the same programming assignments. More than half of the participants were computer science majors, and the rest were electrical engineering majors. Most of the participants were second or third year undergraduate students.

Students entering the Advanced Data Structures and C++ course must first complete a course in C programming and data structures with an introduction to C++ programming. Students are

assumed to be familiar with elementary C/C++ programming concepts, including pointer arithmetic, dynamic memory management, classes, standard library `vector` and `string` classes and the `iostream` library. They are expected to know how to write and debug simple C and C++ programs with multiple source and header files, and to be familiar with the design and implementation of elementary data structures, including arrays, strings vectors, linked lists, binary trees, stacks, and queues.

The Advanced Data Structures and C++ course begins with a review of the fundamentals of C++ programming, C++ development, and debugging in a Unix Environment. Next, object-oriented programming concepts are introduced, including function overloading, object construction, copy control, inheritance, and polymorphism. Finally, advanced data structures like search trees, hash tables, and dictionaries are discussed. Students are presented with many examples of programs written in the object-oriented style, both in lectures and the textbooks. In their weekly programming assignments, students either write programs of their own or modify programs provided by the instructor.

3.7 Material and Tasks

3.7.1 Selection of Tasks

Students were assigned eleven programming tasks, with approximately one each week. Three of the assignments were selected for examination in the pilot: the second, third, and seventh. The selected assignments did not depend on code developed as part of any other assignment.

In the second assignment, students modified a grade computation program from the textbook to reflect the grading policies of the Advanced Data Structures and C++ courses. Their program was expected to read name and grade data from a file or from standard input, and print an alphabetical list of students grouped by letter grade. I will refer to this assignment as the *grading problem*.

In the third assignment, students defined iterator classes to complement a vector class defined in the textbook. The students' iterator classes supported the pointer-like interface of C++ standard

library iterators, as well as a simpler interface consisting of only two member functions, `next` and `hasNext`, described in the textbook [25]. I will refer to this assignment as the *iterator problem*.

In the seventh assignment, students implemented a general tree data structure as a template class, supporting an interface defined in the textbook. This contained a single template argument specifying the type of the elements stored. I will refer to this assignment as the *tree problem*.

In the pilot study, I used the grading problem to identify trends in student debugging behavior, as students debugged a mixture of their own code and code provided to them. The iterator and tree problems were used to study the behavior of students debugging their own code. These assignments might be simple exercises for professional programmers, but for novice programmers they can not be considered simple. Students spent between ten and twenty hours on each assignment.

3.7.2 Tools for Assignment Download and Data Collection

The pilot study was conducted using *Eclipse*, an integrated development environment (IDE), with the C/C++ Development Toolkit (CDT) plug-ins, which provide a powerful set of C/C++ development and debugging tools. The automated logging facilities used to monitor student debugging behavior are integrated into the Eclipse IDE in the form of a plug-in, and they are also available to participants through the web-based installation and update services built into Eclipse.

The logging plug-in monitors debugging actions and the evolution of the program code. In addition to tracking changes in program source files, the following events are logged and time-stamped:

- Launching and terminating the Eclipse IDE.
- Creating, modifying, moving, or deleting a project, file, or folder.
- Opening or closing an Eclipse project.
- Beginning or completing a program build.
- Running a program, or terminating a program that is running.

- Launching or terminating a program in the debugger.
- Setting, disabling, enabling, or clearing breakpoints (logged with breakpoint information such as project name, file name, line number, condition and counts).
- Setting or modifying a watch expression (logged with the value of the expression).
- Stepwise execution of a program in the debugger.

This time-stamped data was gathered automatically while the participants work on their programming assignments, and submitted along with the completed assignments.

3.8 Procedure

3.8.1 Preparation

Before conducting the study, I selected and informed participants as well as training them on the IDE program they would be using.

Abtaining Consent

Students in the Cpts223 class of Fall, 2004 were asked to participate in the study of their debugging practices. Students were briefed on the general purpose of the study and how the data would be collected. Students who agreed to take part were required to sign a written consent form. Students were informed that they would not receive academic credit for participating, and that they could withdraw from the experiment at any time without affecting their course grade.

Training

Students attended 2 one-hMy introductory lectures on programming and debugging C++ programs in Eclipse. Students also accessed the course web-page for a short tutorial on Eclipse. Students also observed a short demonstration on how to use my plug-in to download assignments and prepare the archive for submission. The archive is a compressed file which includes all the source files required in each assignment.

3.8.2 Execution and Data Collection

The study was executed over four months, during which the 11 programming assignments were submitted almost weekly. Three of these assignments were selected as tasks in the study, as described in Section 3.7.1. Data on students' debugging activities and on the evolution of students' programs was gathered automatically by the integrated development environment as participants worked on their assignments as described in Section 3.7.2, and log data was submitted along with completed assignments.

3.9 Analysis

3.9.1 Data Validation

After the data was collected, I evaluated its validity. Factors evaluated included whether participants seemed focused during the experiment and developed their programs in the special version of Eclipse with tools for assignment download and data collection.

Because the activity monitor recorded differences between successive versions of source files rather than complete snapshots, pilot data was sensitive to corruption that would prevent me from reconstructing students' program code. Therefore, I evaluated data for corruption and rejected any data that was compromised for the following reasons:

- The activity log file was corrupt (deliberately or accidentally).
- Participants edited their program code outside of my special version of Eclipse environment, so some changes were not tracked in the activity log.
- The activity log file was missing from the submitted assignment.
- Participants did not run or debug their program in the Eclipse environment, so the timing of debugging events was not recorded.
- Participants dropped the class, or failed to submit an assignment.

Table 3.1: Participants' graded performance.

participant	Course grade	Grading	Iterator	Tree
A	A-	100	80	90
B	B	100	26	100
C	C	90	90	107
D	D	55	26	71.5
E	Fail	35	66	45

Table 3.2: Acronyms used in the data analysis.

IPRT	initial program reading time (in minutes)
TNR	total number of program runs
TND	total number of program runs in the debugger
TNRC	total number of runs for program comprehension with print-out statements in unchanged instructor code

A collection of scripts were developed to automatically identify invalid data. After I evaluated the collected data, five students assignments were considered valid, and these data were analyzed further. The performance of these five participants on the individual assignments and in the course overall is tabulated in Table 3.1. Individual assignment grades are on a 100 point scale. The data of thirty-three students was rejected because I classified it as corrupt.

3.9.2 Data Analysis

The data were analyzed with respect to program comprehension strategies, debugging behavior, and logic errors. I found that participant A, a strong programmer, exhibits some of the qualities associated in the literature with "expert" programmers. This participant's behavior differs from that of the other participants in program comprehension strategies and debugging behavior.

Table 3.2 expands the acronyms used in the presentation of the pilot study data.

Program Comprehension

Debugging statistics for the grading problem are tabulated in Table 3.3. When asked to modify unfamiliar code provided to them, participants B, C, D and E first studied the program intensively before making any changes. This is evident from the initial program reading time (IPRT). This

Table 3.3: Debugging statistics for the grading problem.

Participant	IPRT	TNR	TND	TNRC
A	45m	8	8	0
B	30m	88	58	15
C	25m	82	93	18
D	15m	40	4	5
E	27m	45	20	10

activity did not differ qualitatively from that of participant A. However, from the number of times that participants executed the program modified only by the introduction of a printing statement (so-called "printf debugging"), I can infer that although participants B,C,D and E did study the program before modification, they then shifted to an isolation approach in which they attempted to modify the original program code without fully understanding it. Participant A did not engage in "printf debugging" by printing statements into the original program code provided.

Debugging Behavior

When a program generated output that did not match the programmer's expectations, I observed that participants tried to locate errors by tracing program execution and checking the state of the program variables. But when an error did not produce a failure, the participants did not find the error at all. I propose that enhanced debugging tools that monitor program state, rather than program output, would help programmers find errors that do not produce failures.

Debugging statistics for the iterator problem are tabulated in Table 3.4, and for the tree problem in Table 3.5. From these statistics, I infer that Participant A prefers to use the debugger, while participants B, C, D and E prefer to run the program without the aid of the debugger. It is possible that more training on the use of the Eclipse debugger would encourage more students to take advantage of its facilities.

Participants B,C,D and E made more hypotheses about their errors than participant A, and then made extensive changes in their programs, introducing additional errors, in the course of looking for errors that caused the failures they observed. The additional errors made it even harder to

Table 3.4: Debugging statistics for the iterator problem.

Participant	TNR	TND
A	3	2
B	5	8
C	2	3
D	0	0
E	29	21

Table 3.5: Debugging statistics for the tree problem.

Participant	TNR	TND
A	6	8
B	123	6
C	94	25
D	220	2
E	74	17

isolate the original error.

I found that participants B, C, D and E repeated the same types of defects frequently throughout a program. Moreover, participants B, C, D and E made many small changes and then did lots of testing, whereas participant A made more substantial changes before testing.

Logic Errors

Most visible errors in these assignments were found and corrected in approximately one hour. For example, in the grading problem, participants B and C read character data from a file stream into a character pointer for which no memory had been allocated which is shown in Figure 3.2. Participants D and E put their modifications in the wrong place of the provided code. Participants B and D put their modifications to the provided code, but did not remove the unnecessary program statements that should have been replaced by their modifications.

In the grading problem, common errors all participants like to make are as follows:

- Memory problem: Participant B and C read a char data from fstream to a char pointer, but they did not open memory for the pointer.

```
// This pointer has not been allocated any memory
char *test;

ifstream fileStream( filename );

// Read character data from a file stream into a
// character pointer for which no memory had been allocated
fileStream >> test;
```

Figure 3.2: Memory Problem: Read character data from a file stream into a character pointer for which no memory had been allocated

- Spurious problem: Participant B and D added their implementation to instructor's code, but they did not delete the instructor's unnecessary program statements.

- Misplaced problem: Participant D and E put their implementation to the wrong place of instructor's code.

- Expectation and interpretation problem: Participant D and E misunderstood the program specifications. For example, when the grade records are printed, records are sorted first by letter grade, then by name, but participant D and E' programs just sort grades by name. Moreover participant D and E's programs used the wrong formula to caluate students grade.

In this assignment, almost all errors except expectation and interpretation problems are easy errors in the category of visible error, and students fixed them in an hour. The reason why participants D and E got low grades is because their expectation and interpretation problems prevented them from detecting the apparent failures.

In the iterator problem, common errors all participants are likely to make are as follows:

- Variable bugs: iterator.end() is supposed to return the iterator to the position at the end of vector, but participant C used the wrong variable and made this function return the iterator to the current element position in the vector.

- Unexpected cases problem: iterator.hasNext() must return true if the iterator is not yet at the end of the sequence (that is, if it is not equal to the iterator returned by end()). But participant C and E implemented this function so that if current iterator position added by 1 is not the same as the end position of vector, then the iterator.hasNext() returns true. When the iterator is at end of the vector, the hasnext() function need to return false, but participant C an E did not consider that case and their program will return true.

- Initialization problem: Participant C and E did not initialize current element position pointer, or the end position pointer, or both in Iterator++ and iterator constructor that accepts a vector argument. This makes the runtime state fault, but if the test case is not strong enough, this fault will not cuase a failure.

- Mismatch between modules: Iterators should provide a constructor that accepts a Vec argument, but participant B use his own signature which is different with specification. Participant E used Next() instead of next().

- Expectation and interpretation problem: Participants A and E did not know the specifications about Const Iterator class definitions, so they failed to get the right signature for VecConstIterator.

- Data Structure debacle: At the beginning, participant E did not have the end positon pointer in his iterator class

The reason why participant E got a low grade is because his expectation and interpretation problem prevented him from creating the right signature for VecConsIterator. Participant B did not use the required signature for the constructor which accepts a Vec argument. This caused most of the test cases from the grader to fail in runtime, so participant B got a low score too. Participant D got a low grade because his program had many compiler errors and cannot run at all.

```
// VectorIterator<T> has member variables:
// T* mPosition  (a pointer into the Vector's memory)
// T* mEnd (a pointer to one past the end of
//          the Vector's memory)

// Correct implementation of hasNext:
// Return true if the iterator refers to a
// valid position in the Vector.
Template <class T>
bool VectorIterator<T>::hasNext( void )
{
   return (mPosition != mEnd);
}
```

Figure 3.3: The Correct implementation of `hasNext`. This function return *true* if and only if the iterator is not at the end of the sequence of elements stored in the vector, that is, if its member variable `mPosition` is not equal to its member variable `mEnd`.

In this assignment, the errors of Variable bugs and Data Structure debacle are in the category of visible error, and they have been fixed effeciently with the current debugger in 20 to 60 minutes.

I found that latent and hidden errors went unfixed in the participants' code. Figure 3.3 and Figure 3.4 show two implementations, one correct and one incorrect, of the `hasNext` member function of the iterator class developed in the iterator problem. This function must return *true* if and only if the iterator is not at the end of the sequence of elements stored in the vector, that is, if its member variable `mPosition` is not equal to its member variable `mEnd`.

Participants C and E submitted the incorrect implementation, which returns *false* when the iterator refers to the last valid position in the vector, and *true* when the iterator refers to a position after the last valid position. The vector member function `end` typically returns an iterator to the position one past the last valid position, and so in the incorrect implementation, `hasNext` will incorrectly return *true* when invoked on the iterator marking the end of a vector. Participants C and E did not consider this case when they tested their iterator, and their test programs did not cause this error to generate a fault, so this latent error was unaddressed.

```
// Incorrect implementation of hasNext:
template <class T>
bool VectorIterator<T>::hasNext( void )
{
   //The vector member function end() typically returns
   //an iterator to the position one past the last valid
   //position, and so in the incorrect implementation,
   //hasNext will incorrectly return true when invoked
   //on the iterator marking the end of a vector.
   return ((mPosition +1) != mEnd);
}
```

Figure 3.4: Incorrect implementation of `hasNext`. This incorrect implementation returns *false* when the iterator refers to the last valid position in the vector, and *true* when the iterator refers to a position after the last valid position.

Figure 3.5 and Figure 3.6 show two implementations, one correct and one incorrect, of the post-increment operator for the vector iterator class developed in the iterator problem. The post-increment operator advances the iterator to the next position in the vector, but returns an iterator to the position it held before the advance, that is, it returns a copy of the iterator as it was before the post-increment operator was invoked.

Participants C and E incorrectly initialized the iterator returned by the post-increment operator. Instead of initializing a copy of the iterator, they initialized an iterator in its default state. The participants tested their implementations of the post-increment operator, but only to ensure that the increment operation occurred. They never verified that the correct value was returned from the function, so although their test programs generated faults, these faults did not generate failures, and the hidden error went unnoticed.

In the tree problem, common errors all participants like to make are as follows:

- Data structure debacle: All participants have errors using and changing tree data structure, especially when they implement function copySubtree. The copySubtree method recursively duplicate Nodes in a subtree. Participants had a problem of making the right connections

```
// Correct implementation of post-increment operator:
// Increment, and return a copy of the iterator before
// it was incremented.
template <class T>
VectorIterator<T>
VectorIterator<T>::operator++( int )
{
    VectorIterator<T> ret(*this); // copy
    mPosition++;
    return ret;
}
```

Figure 3.5: Correct implementation of the post-increment operator. The post-increment operator advances the iterator to the next position in the vector, but returns an iterator to the position it held before the advance, that is, it returns a copy of the iterator as it was before the post-increment operator was invoked.

```
// Incorrect implementation of post-increment operator:
template <class T>
VectorIterator<T>
VectorIterator<T>::operator++( int )
{
    // Instead of initializing a copy of the iterator,
    // This incorrect implementation of post-increment
    // operator initialized an iterator in its default state.
    VectorIterator<T> ret;  // default
    mPosition++;

    return ret;
}
```

Figure 3.6: Incorrect implementation of the post-increment operator. This post-increment operator incorrectly initialized the iterator returned by the post-increment operator. Instead of initializing a copy of the iterator, it initialized an iterator in its default state.

between the parent node and child node. The child pointer of the parent node should point to the children and the parent pointer of the child should point to the parent. Participant usually make first case successfully, but failed to make the second case work. Moreover participant E and D made some misktake in the beginning, such as returning a copy of the parent when calling the node.parent().

- Unexpected cases problem: Participants did not consider handling some extreme cases, such as getpath(t,t.root).

- Expectation and interpretation problem: Participants did not implement exception handling for the right cases; participants made the wrong function signature, especially for const object. Participants E and D lacked an inadequate comprehension of the tree.

All of the participants introduced errors in their implementation of the tree data structure, particularly in the implementation of the `copySubtree` function used to perform copy and assignment operations. `copySubtree` is a recursive function that returns a deep copy of the subtree rooted at the node that is its argument.

All participants had difficulty correctly linking parent nodes to child nodes. Adding children to a node was not difficult, since the child node pointers were simply stored in a vector, as indicated in the `Node` structure definition in Figure 3.7. But the parent pointer for each node needed to be explicitly assigned the correct node address. Figure 3.8 and Figure 3.9 shows two implementations, one correct and one incorrect, of the `copySubtree` function. In the incorrect implementation, the parent pointers are assigned to nodes in the original tree, instead of to the corresponding nodes in the copy. Figures 3.10 and 3.11 show the trees constructed by these two implementations of `copySubtree`.

For participants B, C and E, the error in the `copySubtree` problem is in the category of visible errors. Participants B and C spent more than three hours locating and fixing the error. Participant E spent more than 4 hours locating the error, but still did not fix it successfully. For

```
struct Node
{
   // store children in a vector
   vector< Node * > mChildren;

   Node * mParent;
   Object mElement;

   Node( Object obj ) : mParent( 0 ), mElement( obj )
   {
   }
};
```

Figure 3.7: Definition of the `Node` structure used in the tree problem. A Node needs to store an Object (the element stored in that node of the Tree), a pointer to its parent Node, and a vector (or other container) of pointers to child Nodes.

participants A and D, this problem is in the category of hidden errors. These participants verified the links in the copied tree from the root to the leaves, but did not verify the links from the leaves to the root, so they never found the error.

3.10 Results

Since the pilot study involved only 5 participants and 3 programming assignments, it is difficult to draw any definitive conclusions. However, the data obtained by monitoring student debugging activities did give me some insight into the way students develop and debug their object-oriented programs, and into the logic errors they typically encounter.

In the task of modifying unfamiliar code, participant A spent more time understanding the code and consequently developed a more complete mental representation of of the program than did the other novice programmers. Participant A was therefore better able to understand the possible sources of errors in the program. Other novice programmers first attempted a comprehension strategy, and then shifted to the isolation approach, in which they attempted to modify poorly-understood code.

```
// Correct implementation of copySubtree:
// Return a deep copy of the subtree rooted at node.
template< class Object >
typename Tree< Object >::Node *
Tree< Object >::copySubtree( Node * n )
{
   Node * copy = 0;
   if ( n != 0 )
   {
      copy = new Node( n->mElem );
      copy->mParent = 0;
      for ( typename std::vector< Node * >::iterator
            chIter = n->mChildren.begin();
            chIter != n->mChildren.end();
            ++chIter )
      {
         // sanity check  for debugging
         if ( 0 == *chIter )
         {
            throw std::domain_error(
               "copySubtree: found child Node == 0" );
         }
         Node * ch = copySubtree( *chIter );
         ch->mParent = copy;
         copy->mChildren.push_back( ch );
      }
   }
   return copy;
}
```

Figure 3.8: Correct implementation of `copySubtree`. This `copySubtree` function is used to perform copy and assignment operations. `copySubtree` is a recursive function that returns a deep copy of the subtree rooted at the node that is its argument.

```
// Incorrect implementation of copySubtree:
template< class Object >
typename Tree< Object >::Node *
Tree< Object >::copySubtree( Node * n )
{
   Node * copy = 0;
   if ( n != 0 )
   {
      copy = new Node( n->mElem );
      copy->mParent = n->mParent;
      for ( typename std::vector< Node * >::iterator
            chIter = n->mChildren.begin();
            chIter != n->mChildren.end();
            ++chIter )
      {
         // sanity check  for debugging
         if ( 0 == *chIter )
         {
            throw std::domain_error(
               "copySubtree: found child Node == 0" );
         }
         Node * ch = copySubtree( *chIter );
         copy->mChildren.push_back( ch );
      }
   }
   return copy;
}
```

Figure 3.9: Incorrect implementation of `copySubtree`. The parent pointers are assigned to nodes in the original tree, instead of to the corresponding nodes in the copy.

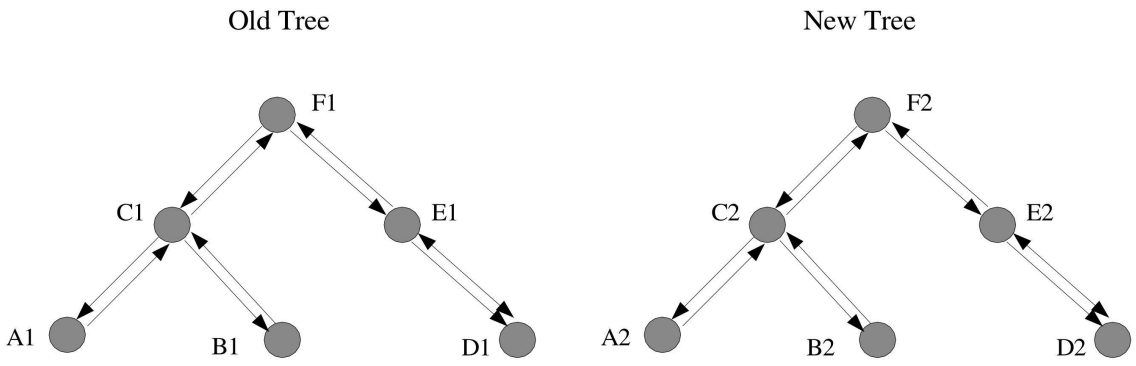

Figure 3.10: A properly-copied tree.

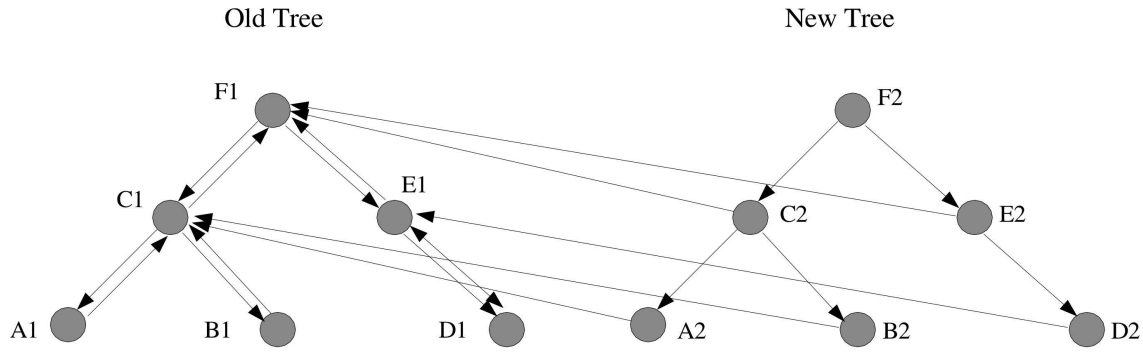

Figure 3.11: An improperly-copied tree, with parent pointers linking nodes in the original tree.

Data structure implementations like the tree problem posed problems for all of the participants. Some spent a lot of time debugging, not always successfully. Others never discovered the errors. In my later proposed experiment, I will show how program auralization can be created to help programmers detect the visible and hidden errors in this chapter.

CHAPTER FOUR

NON-TEXTUAL REPRESENTATION OF PROGRAM STATE

4.1 Introduction

Debugging compiled programs has traditionally been a complex endeavor. In order to reduce errors, programmers must be able to relate several static and dynamic views of a program. Unfortunately, traditional debugging tools are text-based and independent, requiring the user to manually associate the state of one tool with that of others. This in not only inconvenient, but taxes the user's short-term memory.

As computer hardware has become increasingly complex and graphics cards have become standard equipment on personal computers and workstations, graphics display technology has become more sophisticated. The graphical user interface has replaced the text-based display in all but a handful of cases. One of the main areas to benefit from advances in graphical technology has been that of software visualization. Visualization tools assist users in making sense of complex data structures and comprehending the run-time behavior of programs

In this chapter, I will briefly describe some of the attempts to use software visualization to improve program debugging, and I will discuss how the emergence of improved computer audio technology is helping to improve debugging.

4.2 Graphical Debugger

Graphical display transforms information into useful visual representations for programmers. It is the most widely-used and researched alternative to textual representations of program state and behavior.

Blaine Price, Ronald Baecker and lan Small defined software visualization as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer

interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software [58].

In software visualization, mappings are made between software attributes and graphical objects. Such mapping provides a framework to help users construct mental images of the states and structures of the program. Except for animation, most graphics are a kind of spatial media, and software visualization reflects the structure and components of the software. The visualization is simply a metaphor with no direct connection to the actual data structure. However, once students have constructed the mapping, they can form mental images or abstracted versions of the data structure in their minds. In practice, an instructor might ask computer science students to visualize a node being added to the branch of a tree so that they can form mental images and draw representations of the data structure.

Since graphical display technology has properties that facilitate software comprehension or make the process of interacting with computers easier, it is usually concerned with the visualization of actual program code or of the data structures of computer programs in static or dynamic forms. It helps programmers make sense of complex data sets and comprehend the run-time behavior of programs and identify bugs, and generaly helps enhance learning and speed development.

For the purposes of education and debugging, various kinds of graphical debuggers, such as Transparent Prolog Machine [22], APT [10], Explorer [55], ETV [6] , ViMer [9] , PECAN [53] , PROVIDE [42] , Amethyst [7] and DS-Viewer [50] , have been developed. Here I will discuss four graphical tools that employ time-based representation as follows.

4.2.1 VIPS

VIPS [30] (Visualization and Interactive Programming Support) is a graphical debugger for list structures. VIPS uses `DBX` as a back debugger and supports the C programming language. Like several other systems, VIPS can dynamically display linked lists as the program executes. Other graphical debuggers require users to define the type of figures for each data item in advance, but

the VIPS debugging tool does not require that procedure because it can automatically acquire data type information from the program to be debugged and rapidly display data structures. Moreover, VIPS has an additional feature of animation rather than instantaneous re-arrangement. The graph changes shape so that the user can easily comprehend how the linked list has been modified. This exploits the feature of human perception that is tuned change recognition through physical movement. VIPS has facilities for managing scale by enabling the user to magnify areas of a collapsed list. The newer version of VIPS extends the original work by adding multiple levels of browsers and letting users interactively identify sublists of interest. Animation facilities support the visualization of dynamic list operations simultaneously among the various views.

4.2.2 FIELD

FIELD, the Friendly Integrated Environment for Learning and Development [59], is the research project that demonstrated the feasibility of practical integrated graphical programming environments. FIELD supports UNIX-based programming such as C. It provides graphical interfaces to a variety of programming tools and integrating these separate tools into a unified whole.

FIELD provides the familiar graph metaphor to represent relationships between data structures during program execution as well as the static relationships between program modules. This integration enables a user to control and analyze the debugging process from the most appropriate view, and the underlying environment keeps this view coherent.

The display tool of FIELD provides automatic data structure visualization without user interaction and lets the user customize the displays using a visual editor.

It obtains the information for its internal structures by querying the terminal-based debugger such as gdb through the message server.

4.2.3 ZStep 95

ZStep 95 [38] is a reversible, animated source code stepper for Lisp. Zstep 95 was implemented in Macintosh Common Lisp 2.0. It is a prototype, and not a production implementation.

ZStep 95 is similar to FIELD in that static and dynamic views are integrated. But ZStep 95 has a more sophisticated display and user-interaction capabilities. The most dramatic of these capabilities is the ability to step backward through the program. This feature significantly increases the ability of the user to deduce the point of error. For example, traditional debuggers stop before the evaluation of each expression and let the user choose whether or not to view the internal details of the evaluation of the current expression. If programmers attempt to speed up the process by skipping over code, they are likely to miss the precise location of bugs. But because the complete, incrementally generated history of the program execution and its output is retained, ZStep 95 allows users to scroll through the complete history of functions by which the program can be stopped at the first sign of trouble, and execution can be backed up until anomalous data is set.

Both FIELD and ZStep 95 demonstrate interesting features for debugging. However, FIELD is more practical, and a number of IDEs use similar techniques (e.g., Visual C++ and Cafe for Java). Although the features of ZStep 95 resemble those of an ideal debugging environment, ZStep 95 is not a production-grade tool. It only targets Lisp 2.0, and the authors claim that adapting it to C would be challenging. A complete parser and unparser for C syntax would be required and care would have to be taken to assure the C interpreter or compiler kept enough type and run-time information.

4.2.4 DDD

Much of the existing research on visual debugging practice has been conducted using imperative programming languages, such as Pascal, and functional programming languages, such as Lisp. Since object-oriented languages such as C++ and Java are increasingly popular in computer science curricula, I will now discuss DDD, the graphical debugger for C and C++ which supports graphical display.

DDD [68] is a novel graphical front-end debugger for command-line debuggers such as`GDB` and `DBX`. It doesn't actually debug, but instead sends all user commands to an actively running `GDB`

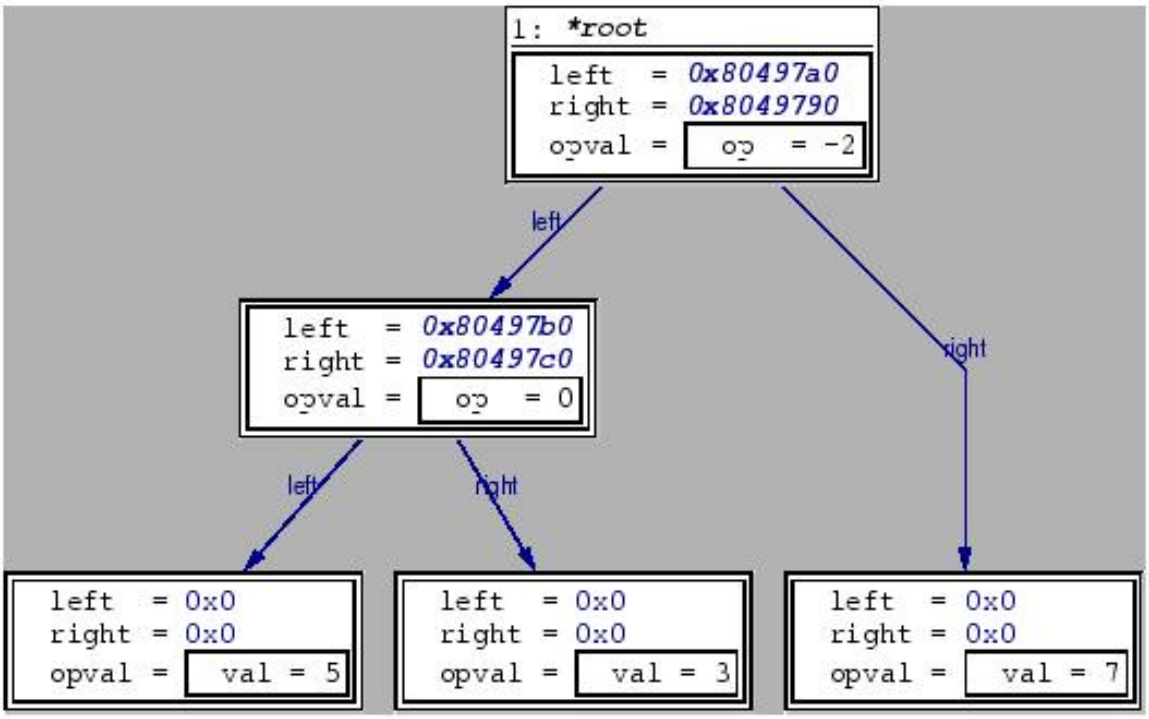

Figure 4.1: Graphical representation of a binary tree in DDD

or DBX process. Besides normal debugging functions such as setting breakpoints and watchpoints, stepping through source files line by line and changing the values of program variables, DDD provides a graphical data display in which data structures are displayed graphically. For data structures without references or pointers, DDD uses boxes to display them. For data structures with references or pointers that cannot be displayed using boxes, DDD uses a graph structure visualizing references as edges between the non-referential data nodes to display them. For example, in a linked list in C, the data structure is a interconnected network, or a sequential collection of self-referential structures connected by nodes. DDD automatically creates diagrams from the memory contents, allowing a simple and appealing visual view to complex structures and allowing users to explored this structure incrementally and interactively. Figure 4.1 shows a graphical display of a tree.

DDD has some drawbacks. It relies on GDB's Command Line Interface (CLI), which has

proved to be very unreliable. Like most visual tools such as Field and Zstep95, DDD was not systematically or empirically evaluated to ascertain its effectiveness.

4.2.5 Lens

The graphical debuggers we mention above are all data structure display systems. But data structure display system can only offer views of raw data and the notion of an overall picture does not exist there. Algorithm animations, conversely, offer views of a programs application domain and semantics. In essence, an algorithm animation is a visual manifestation of a programmers mental model of his or her program. This manifestation presents the program in the way that the programmer conceptualizes it. Consequently, these types of views can be critical for understanding what a program is doing and determining why it is not performing in the desired manner.

Lens [57] which works with the dbx debugger on C programs integrates algorithm animation-style capabilities into a traditional source-level debugger. The primary focus of the system has been to provide these capabilities without requiring a programmer to learn a graphics paradigm and without requiring the programmer to write more code.

4.3 Auditory Display

Much research has been conducted on visualization techniques because they can improve my understanding of software. But it is only during the last decade that research on the use of auditory channels in human computer interaction gained ground. The study of auditory display is dedicated to using sound to present information that has previously been restricted to the visual medium. Recent advances in workstation and PC technology make it easy to generate digital sound, either internally or by interfacing to an external MIDI device, and researchers have found that auditory display can complement, enhance or even surpass the effectiveness of visual representations alone [60]. By 1994, the field had grown to the point that a regular international conference was developed on the topic (the International Conference on Auditory Display). I hope to increase the use of this technology to assist programmers in understanding, debugging, and explaining programs to

others. In this thesis, I am primarily concerned with auditory display.

4.3.1 Sound as an Interaction Medium

With the exception of games and some multimedia applications, sound as a communication medium has been largely unexplored in HCI(Human Computer Interaction). This situation is paralleled in the cognitive sciences by the imbalance between the study of visual and auditory imagery. Most research has been conducted on visual imagery. As user interfaces become more sophisticated, more information can be presented on the monitor screen. This increases the incidence of visual overload, in which the user can no longer take in all the information being presented. The use of auditory imagery promises much in the field of Human Computer Interaction. I will now identify several reasons why sound should be investigated for use as a interaction medium.

- Sound is inherently a temporal medium rather than a spatial one; that is, the time ordering of sound events is what conveys information. Since program execution is a temporal phenomenon, it is logical to explore the possibilities of mapping the program execution to the sound [24].

- Sound does not take up screen space and can fade into the background, but users are alerted when it changes [14] . For example, a driver waiting at a red light might not notice the light turning green until the car behind honks its horn. Most direct manipulation tasks are visual, but visual interfaces are sometimes very crowded and confusing [2, 14]. Vickers notes, “If some of the communication burden can be transferred onto the audio channel, then perhaps interfaces can be made more efficient, more effective, less cluttered and easier to use.” For example, I can use audio to reinforce what is already being displayed visually, or to replace some of that visual display so that users can focus on other visual views.

- Due to the well-known “cocktail party effect”, the ability to selectively attend to one conversation in the midst of others in a crowded room, people can monitor multiple background

process via the audio channel, as long as the sounds representing the processes can be distinguished [14]. For example, programmers do not have to pay strict attention to listening to the normal behavior of a program in order to notice that an exceptional event has happened [24]. Moreover, listening can occur simultaneously with viewing.

- Currently, most interfaces contain an implicit assumption [3] that the recipient has excellent visual capabilities. This is unfortunate, since blind and partially-sighted users are making great strides in using computers successfully. For visually impaired users, exploitation of the auditory channel is essential. For example, the Audiograph [20] system demonstrated that information about graphical objects can be communicated to blind users via musical sequences describing the shape of graphical objects.

4.3.2 Music in Auditory Display

Myriad techniques have been developed to exploit the use of sound in Human Computer Interaction, but there has been almost no research exploring the use of music, which offers a highly structured set of mechanisms for communication. It is surprising that this potential has been largely ignored or underestimated, since music offers unique functionality, summarized as follows [3]:

- People have the ability to identify and recall musical patterns [24]. For example, melodies can be very memorable, and in fact, once learned they are difficult to forget. It is often the case that I hear a tune on the radio in the morning and find ourself still whistling that tune in the evening.

- Music is powerful medium for delivery of large amounts of data in parallel (Even sequential programs can contain an great amount of data) [24, 3]. Its ability to transmit multiple streams in parallel and its ability to transmit in time-based rather than spatial domains offer us new ways of interacting with data. Music involves the simultaneous transmission of a set of complex ideas related over time within an established semantic framework.

- Music offers a highly structured set of mechanism for communicating [2, 3]. Several researchers [16, 23, 48, 61] have proposed that perceived musical structure is internally represented in the form of hierarchies [18]. Since [3] computer programs and many of their associated data structures can be viewed as hierarchical edifices, future research might focus on the use of tonal music to convey hierarchical information in program and data structures.

James L. Alty [1] enumerated some factors which might prevent the use of music in Human Computer Interaction:

- Music is closely linked to local culture, and therefore any musical auralization will have limited appeal.
- Quantitative information cannot be conveyed by sound or music.
- Only trained musicians can appreciate and understand information conveyed musically.

However, many of these arguments can be refuted, as shown below:

- The argument against using music due to cultural basis is an appealing one. But James L. Alty himself has also commented that [1] Western musical structures, whether by independent evolution or cultural imposition [49], have been well absorbed by many of the world's cultures and widely accepted across the world.
- The argument against using music for the representation of quantitative data is harder to refute. Although most individuals can tell if a note increases or decrease in pitch, only trained musicians have the ability to determine exact intervals accurately. However, quantitative information can be meaningfully described and presented in terms of its overall magnitude without the user knowing its exact discrete values [63, 3]. For example, most people can predict ambient temperature by listening to cricket chirps if they know the formula for calculating this.

- The argument that non-musicians will not comprehend musical representation is not reasonable [3]. Many non-musicians enjoy listening to music, whether or not they can carry a tune. On the same token, non-programmers can interact with complex software systems without knowing a program's internal logic and structure. Other researchers, including Walker *ET AL* [4] have shown that musical experience has no effect on participants' ability to interpret auditory mappings of data.

4.3.3 Program Auralization

Auditory display embraces many interesting avenues of inquiry, including program auralization, the topic of this thesis. The term auralization was developed by Brown and Hershberger [8], and typically refers to the process of forming mental images of the behavior, structure and function of computer programs by listening to acoustical representations of their execution. Although sound is not visible, programmers can form auditory imagery of a program by listening to particular sounds or musical pieces representing parts of the program. They can then transform auditory imagery to construct mental images of program, which helps them comprehend the program and communicate with the computer more easily.

The majority of research on program auralization has focused on the auralization of specific algorithms. For example, Brown and Hershberger have shown informally that audio is a powerful tool and medium for conveying information about sorting algorithms [8], and for visualizing the state of parallel algorithms [31, 39, 32]. Several auralization systems, including the InfoSound system [56], LogoMedia [19] , Sonnet system [33], Auditory Domain Specification Language [5]and Listen [17], have been developed.

Musical output may also be used for computer program analysis and debugging [63, 41, 33]. The possibility of using audio for program debugging was first suggested by Francioni [24], who developed a tool to help debug distributed memory parallel programs. The system mapped a particular timbre to each processor and three types of program events. Processor communication,

processor load and processor event flow were represented by unique notes or melodies during execution. The experiments suggest that musical representation can highlight situations which can easily be missed in the visual representation. Francioni concluded that the auralization of parallel program behavior is useful in debugging parallel programs, and software visualization can be enhanced by sound.

Later, Jameson [33] and Vickers [63, 64, 65] successfully described implementations of sonified debugging and code analysis tools called Sonnet and CAITLIN.

Sonnet

The Sonnet system is an audio-enhanced debugger based on its own visual programming language (SVPL). The user tags specific sections of code with auralization objects. Since programmers know what sounds have been associated with what parts of the program, they can predict what sounds will be generated in advance of execution. By listening to the execution of the sound and noticing patterns deviating from the expected path, the user may then investigate more closely the programs where deviation occurs and locate bugs. Sonnet not only allows user to specify how many iterations of a loop to play, but also allow users to monitor and follow program data within a program.

Sonnet runs embedded in a specific debugging environment, and relies on its visual programming languages. Potential drawback is that if a programmer does not explicitly tag a section of code with an agent defining how that section should sound, no sound will be generated and the programmer may not find the bug.

CAITLIN

The CAITLIN system, a non-invasive pre-processor for Pascal that allows users to specify an auralization for each type of program construct (e.g., a WHILE loop or and case statement) [65, 64, 63] has been implemented by Alty and Vickers. This system can determine whether musical feedback can help novices to debug programs. CAITLIN constructs an auditory representation of a program

by associating distinguishing sounds or musical signatures to various language constructs (such as WHILE loops and IF ... ELSE statements). The auralization consists of three basic parts: a signature tune to denote the commencement of the construct, a musical structure representing the execution of the construct, and a signature to signal exit from the construct. The contents of the musical structure within the construct depend on its characteristics. In preliminary experiments, Alty and Vickers showed that programmers, including novices and those with no musical training, can interpret program behavior represented in sound and music. Program auralization helped these novice programmers locate errors more reliable [65]. Most of the problems were caused by participants simply forgetting the tune corresponding to each construct.

Alty and Vickers argue that the inherent durability and memorability of musical melodies, the temporal nature of musical evolution, and the rich perceptual structure of music as a communication medium make it ideally suited to the representation of program behavior. Music involves the simultaneous transmission of a set of complex ideas related over time, within an established semantic framework. The ability to represent many parallel and related data streams organized temporally, rather than spatially, offers new representational possibilities. Moreover, musical representations can effect smooth but precise transitions between contexts when tracking program execution through different program modules, without the disorienting screen reconfigurations of many visual interfaces.

The CAITLIN studies involved only small programs in the imperative style, and it is unclear how the auralization techniques used in those experiments can be scaled to larger and more complex object-oriented programs, or, more precisely, to programs having many features to represent in sound or music. CAITLIN only applied musical auralization to control structures, and the mapping of subroutine calls is not available. Unlike Sonnet, CAITLIN does not allow users to decide which lines or sections of code should be auralized.

4.4 Summary

Most output activity in Human Computer Interaction research has concentrated on the visual medium. Until the last decade, all information provided by the computer had to be graphical or textual in nature in order for interaction to occur between humans and computers. But unfortunately researchers found that students' level of engagement impacts effectiveness more than the visual representaions themselves. Recently, researchers have found that aural representation can complement, enhance, or even surpass visual display, and significant progress has been made in the emerging field of auditory mediums.

Sound provides a kind of extra dimension in visualization. Because listening can occur simultaneously to viewing, it can relieve some of the burden of the visual interface by reinforcing what is being displayed or completely replacing the visual element. Since humans excel at detecting and remembering sound patterns, sound can be used to convey patterns or to alert users of certain events such as signal exception. Since sound has more dimensions than colors, it is an effective medium for delivering large amounts of data in parallel. The temporal nature of sound also makes it ideal for some representations such as program execution.

Sound certainly enhances software visualization to some extent, but how much sound should be used? Although sound conveys patterns well, and is useful in signalling exceptions or alerting programmers to unusual events, users can forget the mapping if too many sounds are presented. It is also more difficult to track exact occurrences in sound, since in the visual format, the user can see a particular state in the history.

Researchers found that sound is more difficult to use than other visual presentation styles such as color, 2D or 3D graphics, and smooth animation. Perhaps this is because people have less training composing music than drawing diagrams. Or perhaps sound has too many parameters such as timbre and frequency. Despite these challenges with using the audio medium effectively in Human Computer Interaction, sound is still a valuable tool for software comprehension and worthy

of further research. In this thesis, I am primarily concerned with auditory displays.

CHAPTER FIVE

AN AURALIZING DEBUGGER

5.1 Introduction

My pilot study and experience suggest that novice programmers tend to develop an understanding of local program behavior, but fail to develop a mental model of the behavior of the program as a whole. This supports Pennington's contention on state information [51, 52] that program state is one of the most difficult aspects of program execution for novice programmers to understand.

Graphical representations of program state have been proposed as an alternative to traditional textual interfaces [38]. The audio channel remains little used in program debugging. The evolution of program state is a *temporal* phenomenon. Sound, on the other hand, is inherently a temporal medium. The mechanisms of auditory perception tend to focus my attention on the dynamic aspects of an auditory scene, and ignore aspects that are unchanging.

It has been suggested that the properties of sound and listening make auditory representations ideally suited to improving program comprehension [24, 2, 14]. Auditory display, though rarely used in program development and debugging applications may therefore be a sensible alternative to graphical representations of evolving program state.

Prior program auralization systems, including Sonnet [33] and CAITLIN [65] were reviewed in chapter 3. However, neither of these systems provided a flexible, extensible toolkit to encourage auditory representations of program state and behavior in program debugging environments.

To determine whether and how auditory representations of program state and execution can help students form a more complete mental model of the evolving state of a program, I developed a prototype auralizing debugger for non-invasive auralization of program state and execution within the sound or musical context. Ultimately, I hope to develop sonified debugging tools that can improve the debugging performance of students in introductory computer programming classes.

In this chapter, I will first describe the techniques used in this project and then the design of my auralizing debugger.

5.2 Technology

5.2.1 IDE

The Eclipse Java Development tool suite is rapidly becoming the integrated development environment (IDE) of choice for many programmers. Many instructors are also starting to use Eclipse as a tool to help with teaching in the classroom and are recommending Eclipse to their students.

Eclipse is an open source, Java-based, universal platform for integrating development tools. By itself, it is simply a framework and a set of services for building a development environment from plug-in components. Because everything in Eclipse is a plug-in, all tool developers have a level playing field for offering extensions to Eclipse and providing a consistent, unified integrated development environment for users. This parity and consistency aren't limited to Java development tools. Although Eclipse is written in the Java language, its use isn't limited to the Java language. For example, plug-ins are available or planned that include support for programming languages such C/C++, COBOL, and Eiffel.

The Eclipse SDK provides a framework for building and integrating debuggers, collectively known as the debug platform. The debug platform defines a set of Java interfaces modeling a set of artifacts and actions common to many debuggers, known as the "debug model". For example, some common debug artifacts are threads, stack frames, variables, and breakpoints; and some common actions are suspending, stepping, resuming, and terminating. The platform does not provide an implementation of a debugger, that is the job of language tool developers. However, the platform does provide a basic debugger user interface (that is, the debug perspective) that can be enhanced with features specific to a particular debugger. The base user interface operates against the "debug model" interfaces, providing views for a program's call stack, variables, breakpoints, watch items, and console I/O, and allows a user to step through source code. The debug platform also provides

a framework for launching applications from within the Eclipse IDE, and a framework to perform source lookup.

5.2.2 C/C++ Development Tools

The C and C++ languages are among the most popular and widely used programming languages in the world, so it's not surprising that the Eclipse Platform provides support for C/C++ development. Because the Eclipse Platform is only a framework for developer tools, it doesn't support C/C++ directly; it uses external plug-ins for support. The CDT Project is working toward providing a fully functional C/C++ Integrated Development Environment (IDE) for the Eclipse Platform. Although the project focus is on Linux, it works in all environments where GNU developer tools are available, including Win32 (Win 95/98/Me/NT/2000/XP), QNX Neutrino, and Solaris platforms.

The CDT is an open source project (licensed under the Common Public License) implemented purely in Java as a set of plug-ins for the Eclipse SDK Platform. These plug-ins add a C/C++ Perspective to the Eclipse Workbench that can now support C/C++ development with a number of views and wizards, along with advanced editing and debugging support.

Eclipse supports Machine Interface (MI)-compatible debuggers through one of its components: the CDT Debug MI plug-in. But what exactly is an MI debugger? Traditionally, third-party GUI debuggers like ddd and `xxGDB` have relied on `GDB`'s Command Line Interface (CLI) when implementing debugging functionality. Unfortunately, that interface has proven to be highly unreliable. `GDB`/MI provides a new machine-oriented interface that is far better suited to programs that want to directly parse `GDB`'s output.

The C/C++ Development Toolkit (CDT) provides a powerful set of plug-ins that can help you develop C/C++ applications with Eclipse. Although the CDT is still under development, users can take advantage of many of its features today.

5.2.3 MPEG-4 Structured Audio

MPEG-4 Structured Audio (MP4-SA) is an ISO/IEC standard that specifies sound not as sampled data, but as a computer program that generates audio when run. MPEG-4 Structured Audio is based on Csound, a widely-used language for sound design and computer music composition, and syntactically similar to the C programming language which is familiar to many application programmers. MP4-SA is different from standards like the MIDI File Format, because it includes not only the notes to play, but the method for turning notes into sound.

MP4-SA combines a powerful language for computing audio (SAOL, pronounced "sail") and a musical score language (SASL, pronounced "sassil") with legacy support for the MIDI format. MP4-SA also defines an efficient encoding of these elements into a binary file format (MP4-SA) suitable for transmission and storage.

SAOL is the central part of the Structured Audio toolset. It is a new software-synthesis language specifically designed for use in MPEG-4. You can think of SAOL as a language for describing synthesizers; a program, or instrument, in SAOL corresponds to the circuits on the inside of a particular hardware synthesizer. SAOL is not based on any particular method of synthesis. It is general and flexible enough that any known method of synthesis can be described in SAOL. Researchers have written examples of FM synthesis, physical-modeling synthesis, sampling synthesis, granular synthesis, subtractive synthesis, FOF synthesis, and hybrids of all of these in SAOL. If the instrument models use algorithmic synthesis instead of wavetables, an MP4-SA file can describe realistic musical performances without using any audio data – just score data, mixdown cues, and DSP algorithms.

SASL is a very simple language created for MPEG-4 to control the synthesizers specified by SAOL instruments. A SASL program, or score, contains instructions that tell SAOL what notes to play, how loud to play them, what tempo to play them at, how long they last, and how to control them (vary them while they're playing). SASL is like MIDI in some ways, but doesn't suffer from

MIDI's restrictions on temporal resolution or bandwidth. It also has a more sophisticated controller structure than MIDI; since in SAOL, you can write controllers to do anything, you need to be able to flexibly control them in SASL. SASL is simpler than many other score protocols. It doesn't have any facilities for looping, sections, repeats, expression evaluation, or some other things.

All in all,Structured Audio represents a combination of expressive power and familiarity unmatched by other sound synthesis or sound control languages. Structured Audio will be adopted as a language for sound design, to enable experimentation with a sound synthesis algorithms, instrument designs, and orchestration.

5.3 Design

My prototype auralizing debugger is targeted at experimentation rather than application development. It provides facilities to the researcher for experimenting with a variety of mappings between program state and sound, and a platform for testing and comparing the utility and efficiency of auditory representations.

In order to be compatible with conventional debugging environments and to support stepwise execution, breakpoints, data inspection, and expression evaluation, my enhanced debugger renders the program auralization in real time as the program executes. This resembles the way that some other debugging and monitoring tools represent features of a program's state graphically as the program executes.

On modern computer workstations, large sections of a program can be executed much faster than a meaningful auditory representation can be perceived. Therefore, my auralizing debugger regulates the pace of program execution as it is being debugged to ensure the musicality of the auralization.

On the other hand, programs with a concurrent visual display are expected to have the sound and visual streams convincingly synchronized in order to avoid confusion. If the two media are not correctly synchronized, many of the advantages of using sound disappear [66]. My tool can

provide this kind of synchronization between sound and visual display by controlling the update events of all visual displays.

The prototype auralizing debugger was implemented in Java as a plug-in to the Eclipse 3.0 IDE with CDT 2.0. All initial testing of my tool was performed on a Linux machine with the Red Hat 9.0 distribution installed.

The prototype auralizing debugger consists of five parts: the *sonified breakpoint*, the *new CDT debugger engine*, the *instrument module*, the *mapping workspace* and the *event storage* module.

5.3.1 Sonified Breakpoint

In a conventional debugging environment, a line breakpoint is set to suspend the execution of a program so that the programmer can inspect the state of the program when it reaches a certain line of code. Similarly, a special breakpoint could be introduced in a sonified debugging tool to trigger a musical event when the program reaches a certain line of code. My design will start with this kind of special breakpoint.

A *sonified breakpoint* has all the properties of a regular line breakpoint, but differs in that it does not suspend the program. Components in the visual interface, like the variable view (The Variable view displays the object and local variables available at the current position in the execution stack), are not redrawn, and the programmer cannot inspect the state of local variables at the location of the sonified breakpoint. Instead, an event will be sent to the real-time synthesizer to trigger some music events. The program execution continues until next the breakpoint is hit.

Since Eclipse provides client with a chance of seamless integration to Eclipse, users of CDT can add sonified breakpoints by taking the same steps as adding regular line breakpoints through the pop-up menu of the editor ruler: Right-click in the editor or on the marker bar (the vertical bar to the left of the main text area) directly to the left of the line where you want to add the breakpoint and select “Toggle Sonified Breakpoint” from the pop-up menu. Properties of sonified breakpoints, such as the status, condition and ignored count can be set up by using the same interface of setting

properties as for a regular line breakpoint. Figure 5.1 shows the interface of adding a sonified breakpoint.

Eclipse provides a default breakpoint view that shows all line breakpoints, method breakpoints and watchpoints defined in theworkspace. In order to better manage different kinds of breakpoints, I added filters that enable users to easily separate breakpoints by their categories.

5.3.2 New CDT Debugger Engine

In order to implement my sonified tool as a CDT-independent plug-in, I first examine the extensions that CDT provides and the architecture of CDT.

CDT implementation has four layers as follows:

Eclipse Debug UI

The eclipse debug user interface provides a set of classes and interfaces to support a language independent debugger user interface. A generic debug perspective is provided with a common set of views such as debug view, variables view, breakpoint view, expression view, and launch configuration dialog. Clients contribute actions to the debug views via the standard workbench extension points.

Eclipse Debug Core

Eclipse debug core provides classes and interfaces to support facilities common among many debug architectures: launching programs, breakpoint management, expression management, and debug events. An extensible set of debug architectures and languages are supported by the definition of a “debug model” - a set of interfaces representing common artifacts in debuggable programs. The debug plug-in itself does not provide an implementation of a “debug model”. It is intended that third parties providing an integrated set of development tools for a specific language will also implement a “debug model” for that language, using an underlying debug architecture of their choice.

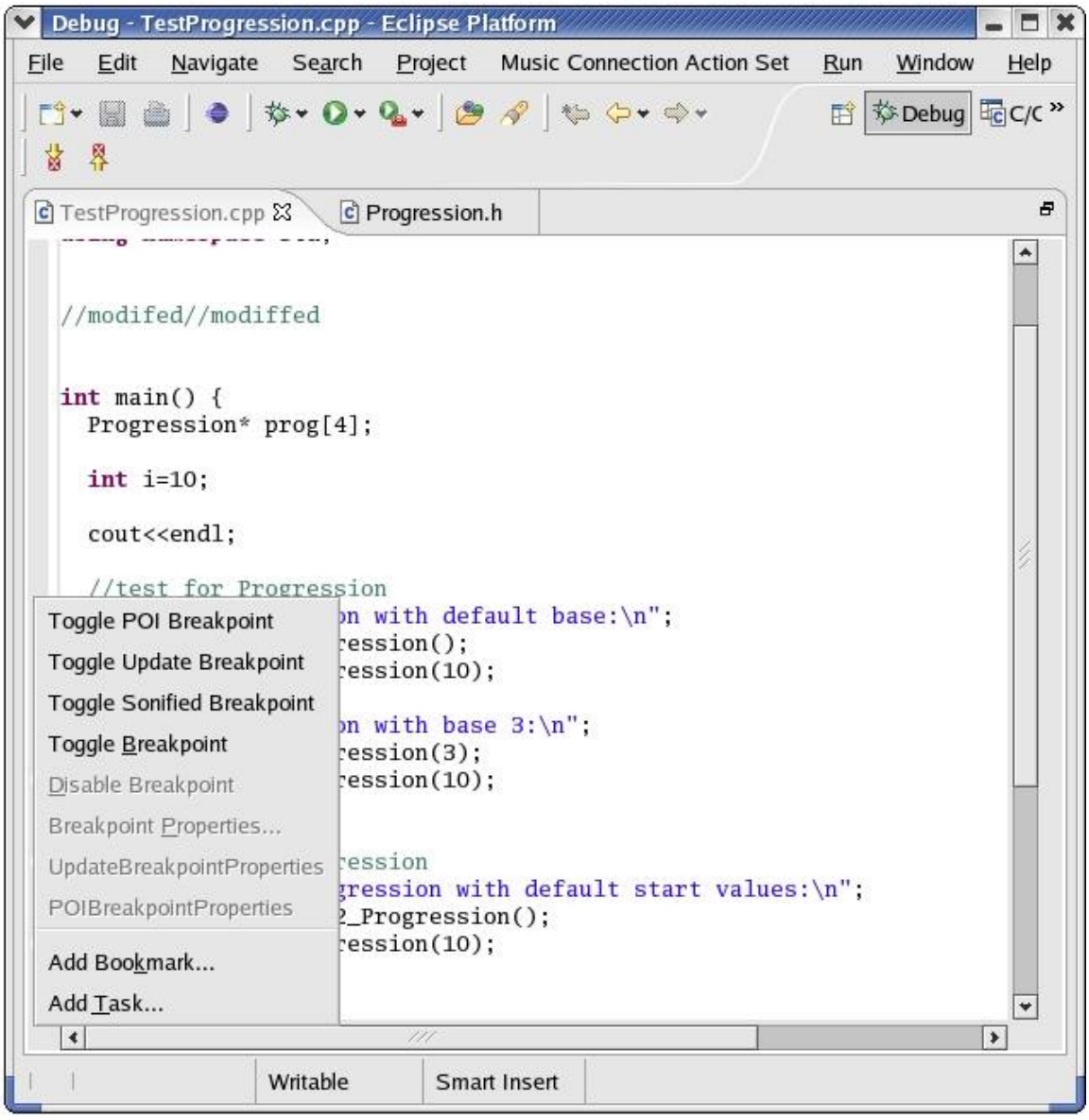

Figure 5.1: Popup menu for adding the sonified breakpoint. After right clicking in the editor or on the marker bar (the vertical bar to the left of the main text area) directly to the left of the line where you want to add the breakpoint, you will see this popup menu, and then you can select "Toggle Sonified Breakpoint" to set a sonified breakpoint.

C/C++ Debugging Interface

CDT provides a layered interface which is a C/C++ specific extension to the eclipse generalized debugger interface. The goal of this interface is to provide a debugger-independent generalized interface by which higher level components such as a UML debugger would be able to interact with a debug session.

`GDB` *Debugger via MI and Super Flexible User Launch Configuration*

Since it is generally useless to have a C/C++ development without some sort of debugger, a default implementation was provided using `GDB`. Of course this isn't very useful without a way to launch `GDB`, so a simple and generic launch configuration was created and integrated into Eclipse's launch model.

After exploring the extensions the CDT provides, I found that only the launch configuration has an extension point. That point is not useful for me, and so there is no way to build my sonified tool as CDT-independent plug-ins. Another factor which prevents me making my tool CDT-independent plug-in is that the current implementation of the CDT debugger does not provide support for the special breakpoint such as sonified breakpoint, and this means I have to provide a new CDT debugger engine. Considering that CDI insulates from the back-end debugger, I think I should not have any reason to poke around this depend layer, which may change from debugger to debugger and even from one different version of MI to another. So I have implemented a patch for the CDT debugger engine to control the pace of program execution and fire up update events of all visual displays in term of filtering on the side of the C/C++ debugging interface.

5.3.3 Instrument Module

The instrument module I created includes both the sound synthesis engine and the real-time instrument driver for receiving messages from the debugger. The sound synthesis engine is based on the Structured Audio Orchestra Language (SAOL) [54]. The sound synthesis algorithms are specified in a SAOL orchestra file and processed using a modified version of the `sfront` tool to build a

dynamically-loadable library implementing the real-time synthesizer. When a sonified breakpoint is hit, the debugger sends a message to the real-time instrument driver, and the driver passes the message on to the synthesizer to play the desired sound or music. Next, I will discuss `sfront` and the real-time instrument driver.

`sfront` *and Real-time Instrument Driver*

`sfront` is a structured audio development tool that processes SAOL,SASL,MPEG-4, and MIDI files to produce C source code(an MP4-SA file). This source code is compiled into an application that, according to the parameters passed to `sfront`, produces samples files, compressed MPEG-4 files and it can play in real time too.

Some end-user applications, such as interactive music production systems use knobs and sliders that are interactively linked to SAOL program variables. External input from a MIDI keyboard must be dynamically routed to different SAOL elements. For `sfront` to support these programs, application-specific code must be added to the MP4-SA file that `sfront` produces. The `sfront` control driver interface lets application developers add C or C++ code into the MP4-SA file file in a standardized way. Control drivers can insert SASL and MIDI commands into a running Structured Audio engine in real time. The control driver API also includes data structures to inform the control driver about the SAOL program structure, as well as functions for reading variables and altering time. `sfront` can, therefore, insert real-time sound synthesis and scheduling instructions into the C source code, and can also insert support for real-time interactivity, using, for example, a hardware MIDI interface.

In my case, I needed to implement a real-time instrument control driver that could receive SASL messages from Eclipse and control structured audio orchestras in real time. Unfortunately, `sfront` does not work like a dynamic library, and the main() function is in the MP4-SA file. I solved the problem by taking the following steps:

- By using the control driver API, I implemented a control driver that can insert SASL and

MIDI commands into a running structured audio engine in real time, and allow the client to access SAOL data structures such as SAOL Instruments describing the data structures that list all the instruments in a SAOL program.

- I created a version of `sfront` that generated no main function MP4-SA file, where MP4-SA file's computing engine is driven via a simple three-functions interface.

- I built an improved MP4-SA file as a shared library, which can function as native implementation for a JAVA.

- I implemented `SAOrchestra`, the Java interface to my shared library version of `sfront`. `SAOrchestra` is a completely encapsulated, low-level Java interface to John Lazzaro's `sfront`. From Java, SAOrchestra just looks like a Java class that translates SAOL files to a single C source code file of non main function form by using my shared library version of `sfront` and shelling out to the operating system to run a C compiler that translates the C file to an executable shared library. The C compiler, library builder and my version of `sfront` are hidden away inside SAOrchestra. The shared library, however, is loaded and managed by `SAOrchesta`. After loading the library, `SAOrchestra` starts a new thread to prepare and launch the real-time instrument driver by calling the Java interface of the three-functions interface in the shared library. My application-specific code in the MP4-SA file program handles the dynamic functions by communicating to the host program `SAOrchestra`

In order to have better control of sending a SASL or MIDI message to a specified instrument, I also implemented the classess of `SaslInstrument`, `MidiInstrument`, `saslmessage` and `midimessage` in my Java interface.

The current implementation of my control driver and `SAOrchestra` provides support to the SASL instr command, creating a new instance of an SAOL instrument. The SASL control command changes the value of a SAOL variable.

5.3.4 Mapping Workspace

The mapping from program state onto parameters of sound is controlled by the *mapping workspace*. The programmer assigns an instrument to each sonified breakpoint. Instruments can range from simple sine waveforms to physical models like bells, and each corresponds to one or more synthesis algorithms that render sound in real time as the program is running in the debugger.

The parameters of the selected instrument are mapped to expressions evaluated in the debugger each time the sonified breakpoint is hit. Instrument selection and parameter mappings can be changed at any time, even while the program is running.

The Expression Sound Mapping View provided by mapping workspace as the interface of setting instrument information for sonified breakpoints is shown in Figure 5.2. It only shows the sonified breakpoints defined in the workspace. The first column indicates whether the breakpoint has been selected for batch setup. The second column indicates the status of breakpoints(enable or disable). The third column displays the breakpoint's location(file path and line number). The fourth column is used for selecting instruments. The other columns are used for the setup of sound parameters.

5.3.5 Event Storage Module

The event storage module communicates with the synthesis instrument. Each time the CDT debugger detects the event of a sonified breakpoint hit, it sends a special event to the event storage module. The event storage module looks up the sound mappings for that type of event. If one exists, the event storage module create and transmits a corresponding audio message to the real-time instrument driver, and the real-time instrument driver asks the underlying synthesis tool to play the corresponding sound. The audio messages used in my tool are based on the SASL format, which is defined in `sfront`. After the sound is played, the event view alerts the CDT debugger, and the program execution resumes.

To solve short term memory problem of recalling melody, I added extra saving and play-back

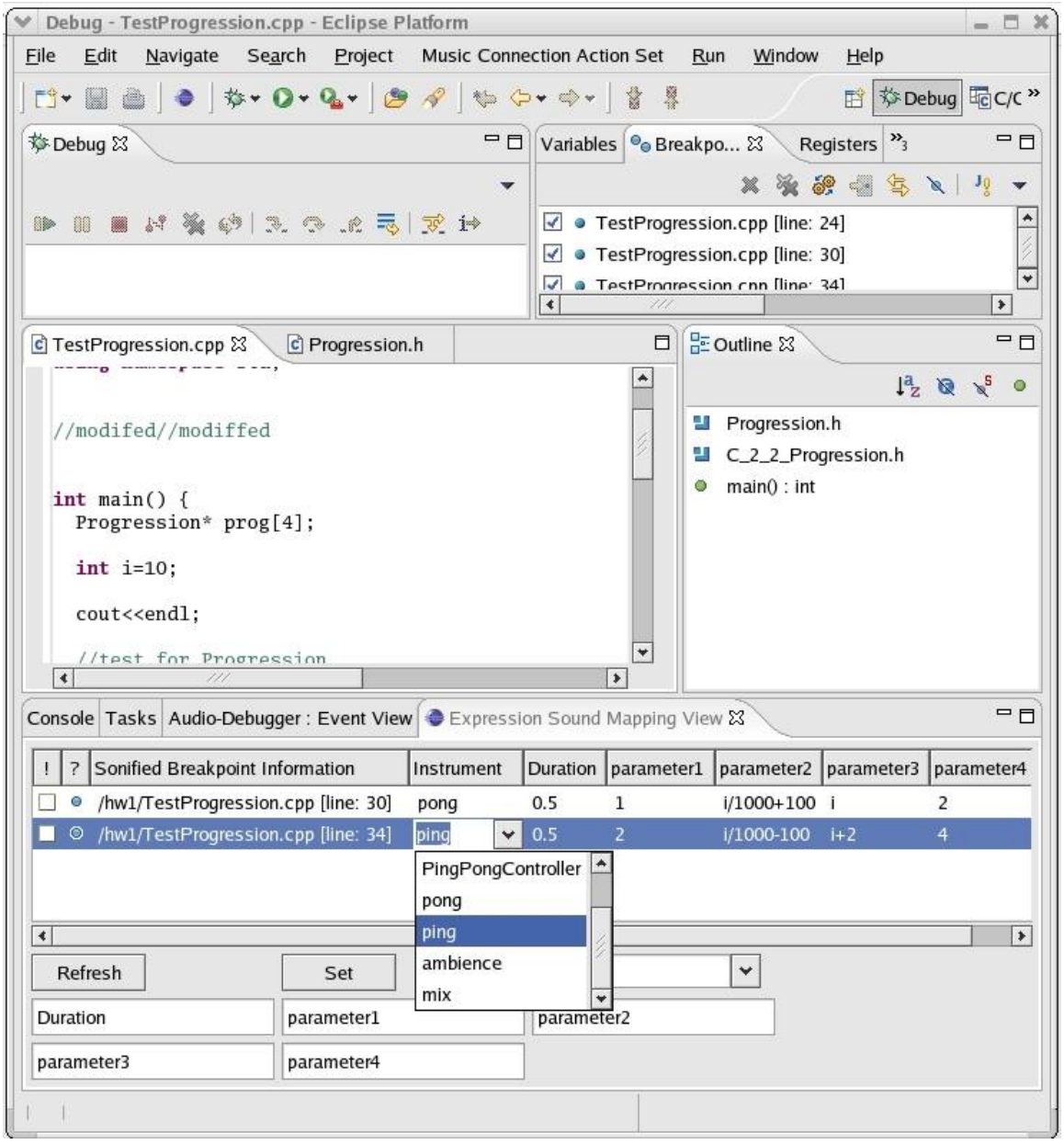

Figure 5.2: The Expression Sound Mapping View is provided by mapping workspace as the interface of setting instrument information for sonified breakpoints. The first column indicates whether the breakpoint has been selected for batch setup. The second column indicates the status of breakpoints(enable or disable). The third column displays the breakpoint's location(file path and line number). The fourth column is used for selecting instruments. The other columns are used for the setup of sound parameters.

functions. The event view in event storage module shows all the events of a sonified breakpoint hit with breakpoint location information. The user can click on one event with the mouse to play the corresponding sound, play all corresponding sounds of all events in a sequence of program execution, save sound information as a sasl file and MPEG4-SA file, or play back the executable MPEG4-SA file. The user can also remove events from the event view that do not pertain to the current debugging task. The event view provided by event storage module is shown in Figure 5.3.

5.3.6 UML Sequence Diagrams on How my Tool Works

To give later developers a better understanding of how my audio enhanced debugger works, UML sequence diagrams are provided to show the interactions between roles in the sequential order that these interactions occur. Note that, here, the lifeline names represent a particular kind of instance (i.e., a role), and not the specific instance of a class.

Adding, Removing, and Changing Regular Properties of Sonified Breakpoints

When a sonified breakpoint is added by user, my tool creates the breakpoint and corresponding marker, and then adds the breakpoint to the `EclipseBreakpointManager`. The `EclipseBreakpointManager` then sends `EclipseBreakpointAddEvent` asynchronously to `CDTDebugModel`. The `CDTDebugModel` adds the breakpoint to the `CDIBreakpointManager` of the CDI layer and the `MIBreakpointManager` of the MI layer. Other actions such as removing or changing the regular properties of the breakpoint require the same steps. Figure 5.4 illustrates how the sonified breakpoint is added, removed and changed.

Start EventView

When Event View is opened, it first creates the instance of `SAOrchestra`. `SAOrchestra` then loads the shared library and creates the real-time instrument driver and synthesizer. When creating the event view, the design of the event storage module typically consists of a model, view, content provider, and label provider. The model will contain the data of the sonifiedbreakpoint hit event I

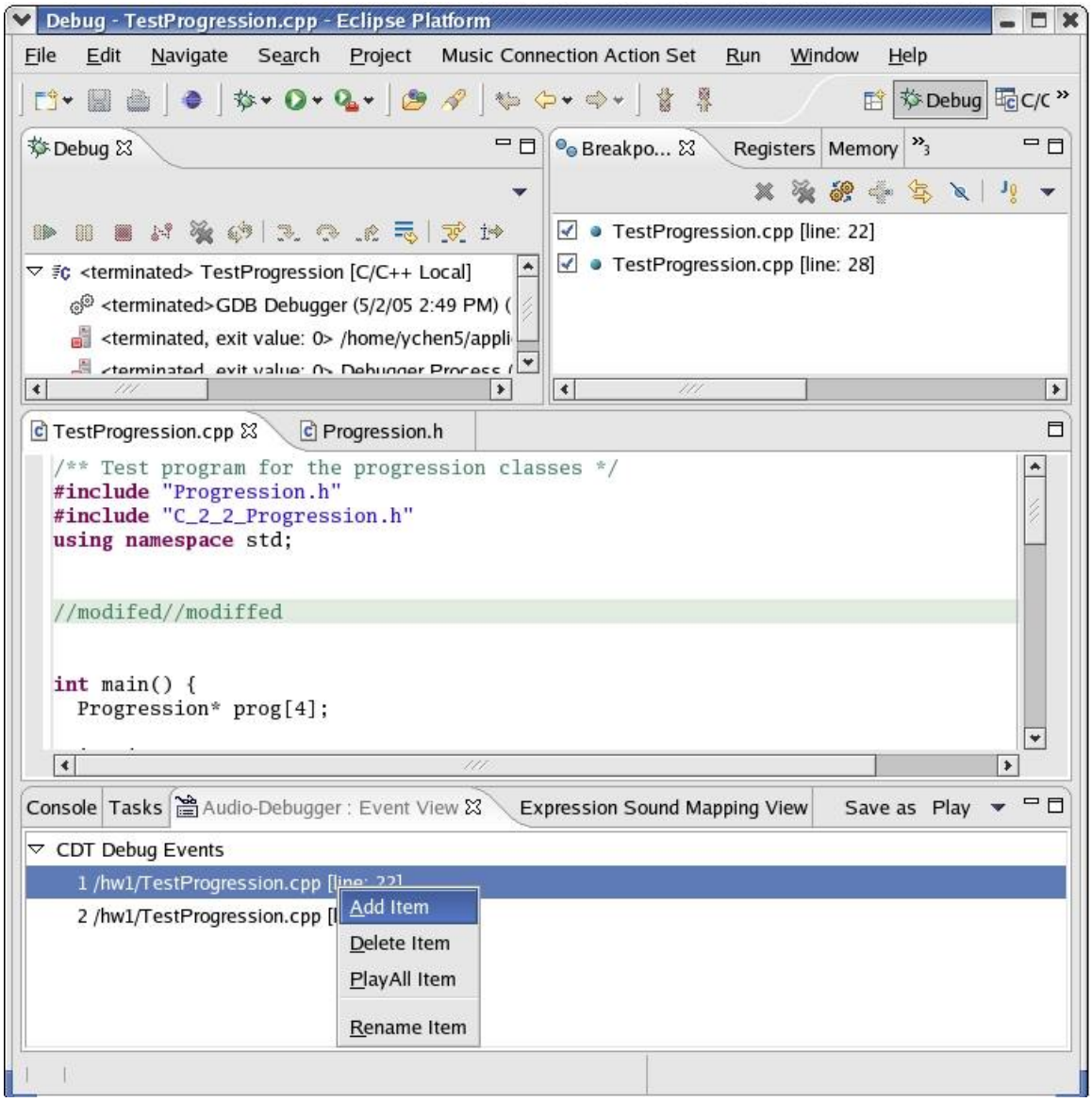

Figure 5.3: The event view provided by event storage module. The event view in event storage module shows all the events of a sonified breakpoint hit with breakpoint location information. The user can click on one event with the mouse to play the corresponding sound, play all corresponding sounds of all events in a sequence of program execution, save sound information as a sasl file and MPEG4-SA file, or play back the executable MPEG4-SA file. The user can also remove events from the event view that do not pertain to the current debugging task.

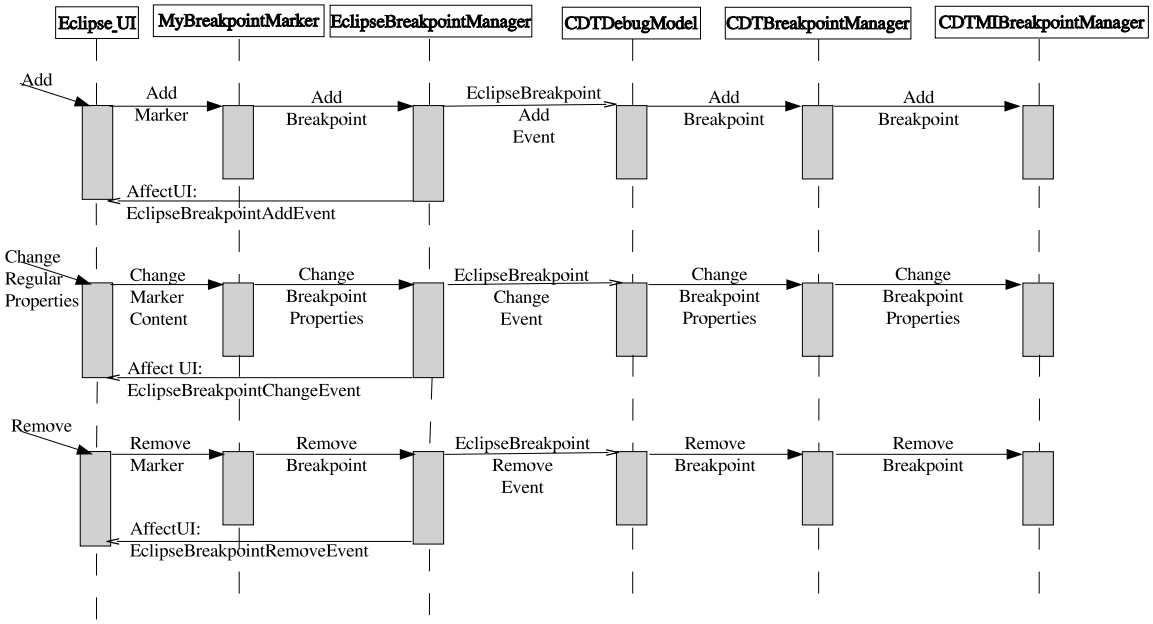

Figure 5.4: Adding, Removing, amd Changing Regular Properties of Sonified Breakpoints. When a sonified breakpoint is added by the user, my tool creates the breakpoint and corresponding marker, and then adds the breakpoint to the `EclipseBreakpointManager`. The `EclipseBreakpointManager` then sends `EclipseBreakpointAddEvent` asynchronously to `CDTDebugModel`. The `CDTDebugModel` adds the breakpoint to the `CDIBreakpointManager` of the CDI layer and the `MIBreakpointManager` of the MI layer. Other actions such as removing or changing the regular properties of the breakpoint require the same steps.

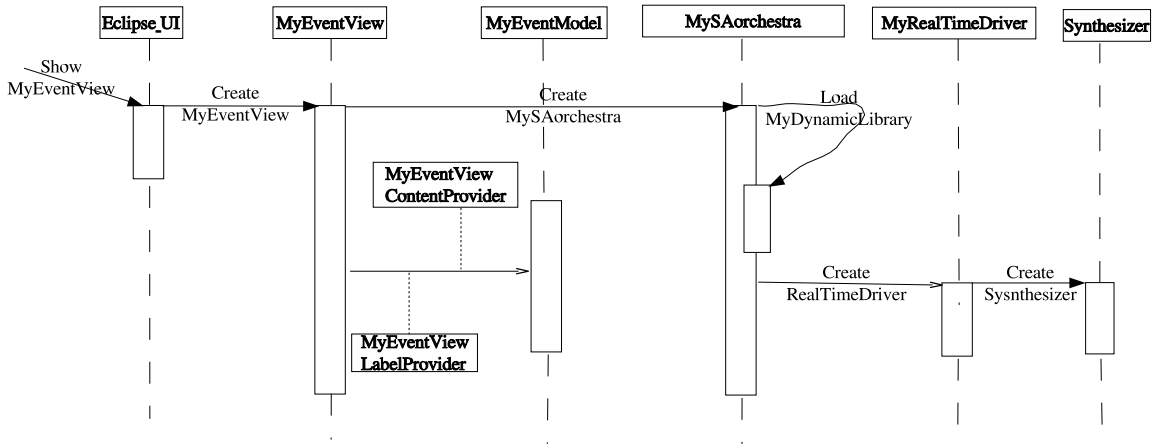

Figure 5.5: Event View. When Event View is being open, it first creates the instance of SAOrchestra. SAOrchestra then loads the shared library and creates the real-time instrument driver and synthesizer.

want to display in the view. It contains methods that access the data and methods that signal when the data has changed. The view represents what the user sees. The content provider defines how the content of the view is obtained from the model. The label provider for the view is responsible for showing a string and icon representation on each data element in the view's control. The model is separated from the user interface since other plug-ins may need to access the data. Thus, I avoid the dependency circle of plug-ins. Figure 5.5 shows how the event view starts up.

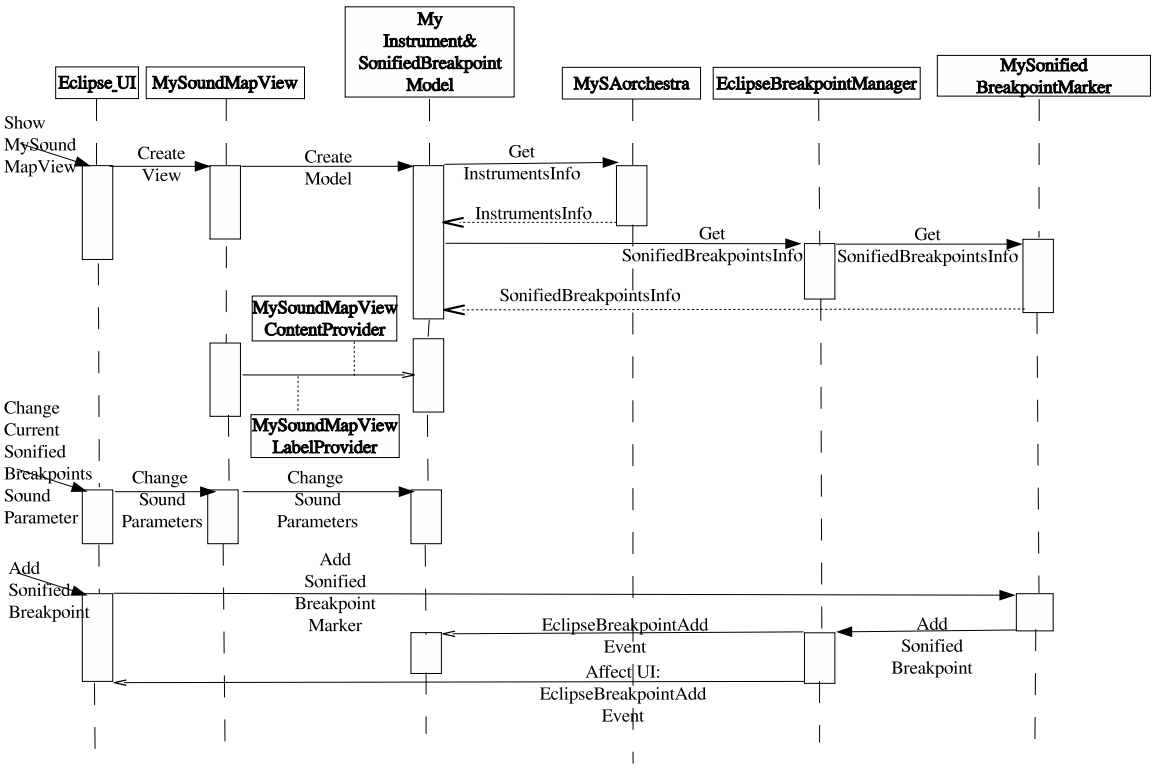

Figure 5.6: Start Expression Sound Map View. Expression Sound Map View has its own model, content provider and label provider. The model contains instrument information such as name and number of parameters, as well as the current sonified breakpoints' information. The model is registered the `EclipseBreakpointListener`, so that when the sonifed breakpoint is added or removed, the model and view will update themselves to reflect the change.

Start Expression Sound Map View

Like Event View, Expression Sound Map View has its own model, content provider and label provider. The model contains instrument information such as the name and number of parameters, as well as the current sonified breakpoints' information. The model is registered as the `EclipseBreakpointListener`, so that when the sonifed breakpoint is added or removed, the model and view will update themselves to reflect the change. Figure 5.6 shows how the Expression Sound Map View starts up.

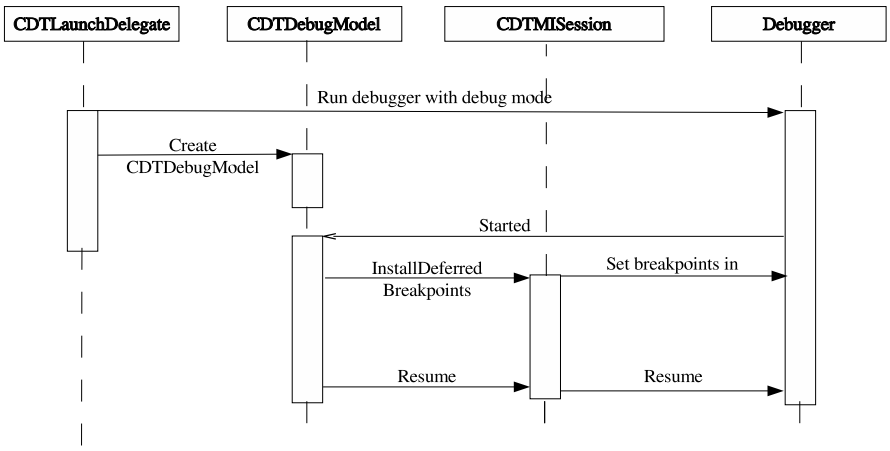

Figure 5.7: Launch Debugger. When the debugger starts up, but before it processes any instructions, the debug target has to reach in and set the initial breakpoints. These initial breakpoints are known as deferred breakpoints.

Launch Debugger Session

When the debugger starts up, it has no breakpoints in CDT breakpoint manager. Users, having set breakpoints in the code using the Eclipse user interface, expect those breakpoints to work. Thus, after the debugger starts up, before it processes any instructions, the debug target sets the initial breakpoints. These initial breakpoints are known as deferred breakpoints. The standard way this initial setup is accomplished is by having the debugger suspend on startup, wait for the debug target to set all the breakpoints, and then have the debug target resume the debugger. The Figure 5.7 shows how the “debug model”, the debugger, and the events interact:

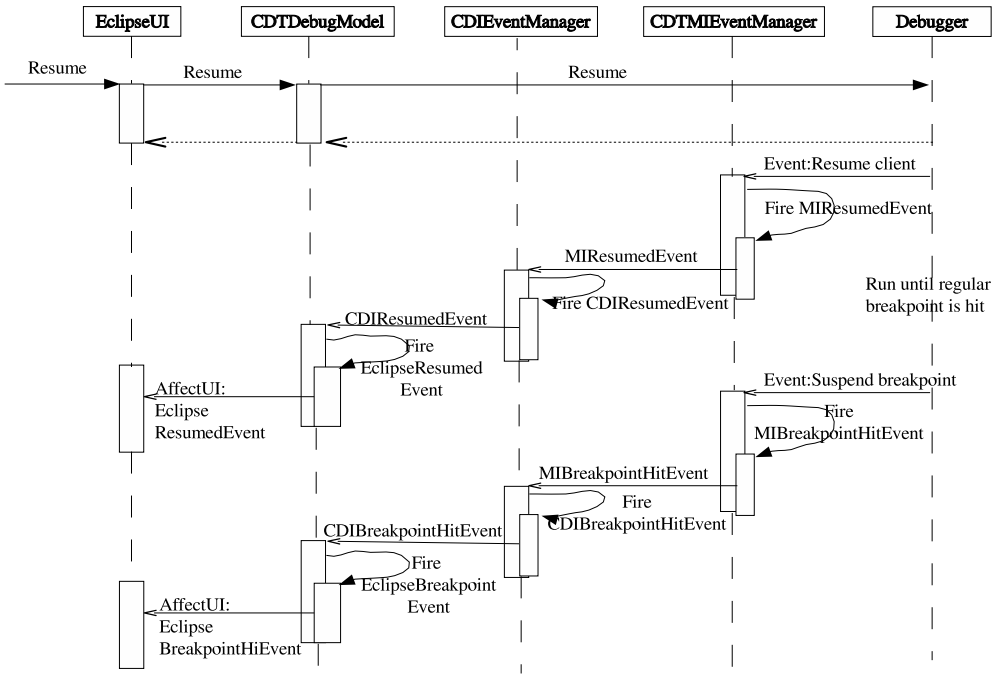

Figure 5.8: This graph shows how the events arrive asynchronously and are processed by the CDT. A resume action from the user interface is sent to the debugger as a resume command on the command channel. The debugger operates asynchronously. It sends the `ResumedEvent` back to `EclipseUI` by way of the event channel. After the debugger runs for a while and hits a breakpoint, the debugger sends the BreakpointHit Event to update the entire UserInterface, such as the `VariableView`

Regular Breakpoint Hit

Figure 5.8 shows how the events arrive asynchronously and are processed by the CDT. A resume action from the user interface is sent to the debugger as a resume command on the command channel. The debugger operates asynchronously by sending the `ResumedEvent` back to `EclipseUI` by way of the event channel. After the debugger runs for a while and hits a breakpoint, the debugger sends the BreakpointHit Event to update the other views in the user interface, such as the `VariableView`.

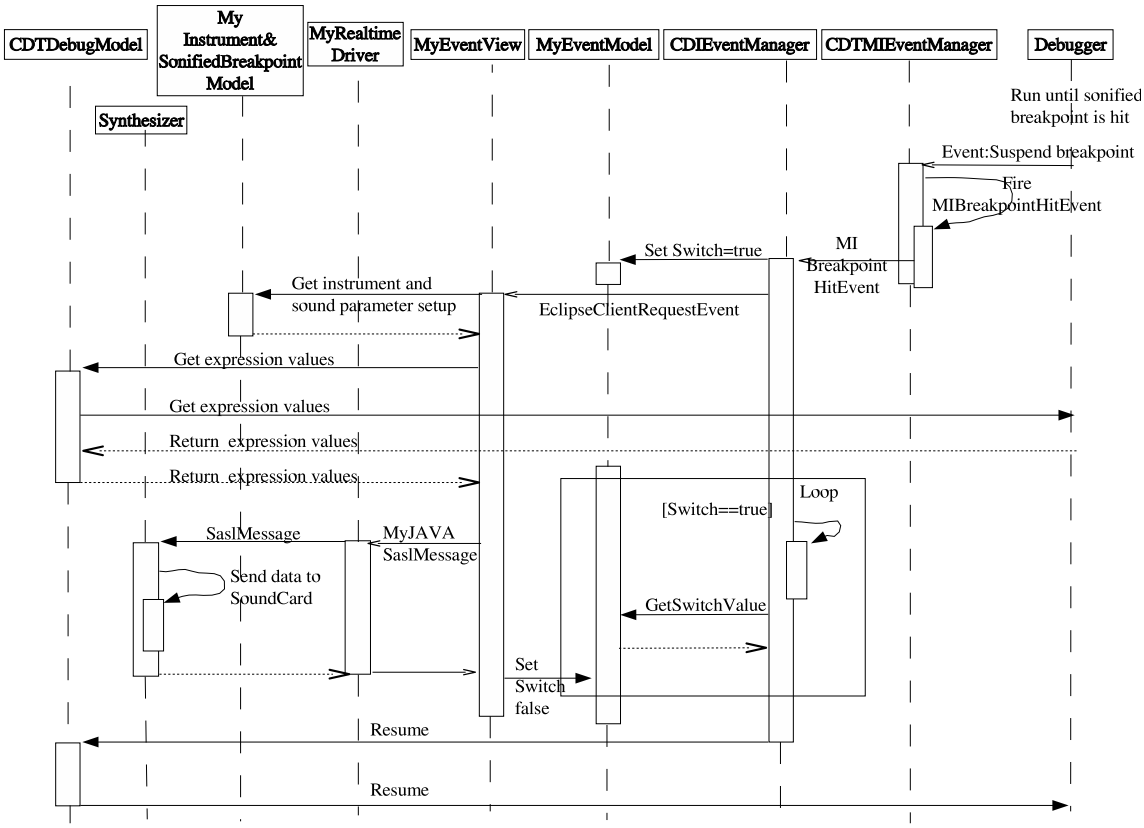

Figure 5.9: This graph shows how my version of CDT process events as they arrive asynchronously. When the sonified breakpoint is hit, all UI components do not update themselves, but instead play the sound. After the sound is played, the debugger resumes and keeps running until the next breakpoint is hit.

Sonified Breakpoint Hit

The Figure 5.9 shows how my version of CDT processes the events as they arrive asynchronously. When the sonified breakpoint is hit, all user interface components do not update themselves, but instead play the sound. After the sound is played, the debugger resumes and keeps running until the next breakpoint is hit.

5.3.7 Example of using sonified breakpoint

Here, I will give an musical representation of program state for detecting a visible error using sonified breakpoint. The iterator post-increment error which was described in the pilot study as a

```
void printRange( Vector< string >::iterator pos,
                 Vector< string >::iterator end )
{
   while ( pos != end )
   {
       cout << *(pos++) << endl;
   }
}
```

Figure 5.10: Iterator post-increment test code.

hidden error will be used as visible error. I will provide a good test program which will cause the program to generate incorrect output. Figure 5.10 shows the test program for sequential access to the elements of a vector using iterators.

In this example, I could do the mapping in the following way. Lines in which iterators were dereferenced were marked with sonified breakpoints. The auralization was based on the addresses of the vector memory and the address of the iterator. Each dereference is auralized by a single chord over two beats (one measure). A valid iterator dereference was represented by a diatonic triad (a three note chord) in four voices, such as the one shown in Figure 5.11A.

Invalid dereference (access outside the bounds of the vector memory) is represented by a dissonance. The note in the alto (second highest) line is changed to produce a dissonance with the note in the tenor (second lowest) line. This is shown in Figure 5.11B. The rhythm is also changed to interrupt the flow in the case of an invalid dereference.

Access to the first element in the vector might be considered a special case, and so in this example, access to the first element is represented by a block chord (all notes played at the same time), as shown in Figure 5.11C, whereas other valid acceesses are represented by arpeggios (notes added to the chord at different times), as shown in Figure 5.11A. Similarly, access to the position one past the end of the vector memory is a special (common) case of invalid access, and could be represented by a modified version of the dissonant chord, as shown in Figure 5.11D.

Figure 5.12 shows the chord sequence for the auralization of iterator. Figure 5.13 shows the

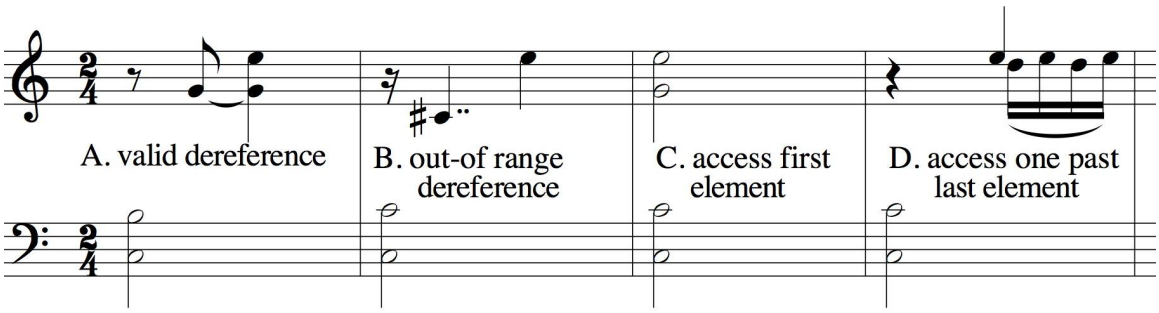

Figure 5.11: Musical signatures used to auralize iterator state.

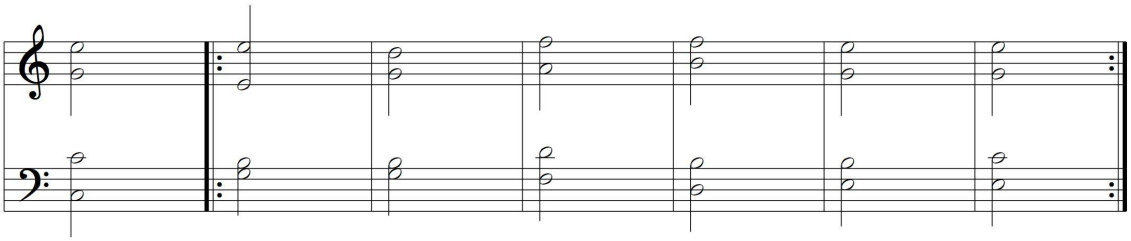

Figure 5.12: Chord sequence for the auralization of iterator.

score for the auralization of the beginning of this iteration when the iterator is implemented correctly. Figure 5.14 shows the score for the auralization of the same iteration when the iterator is implemented incorrectly.

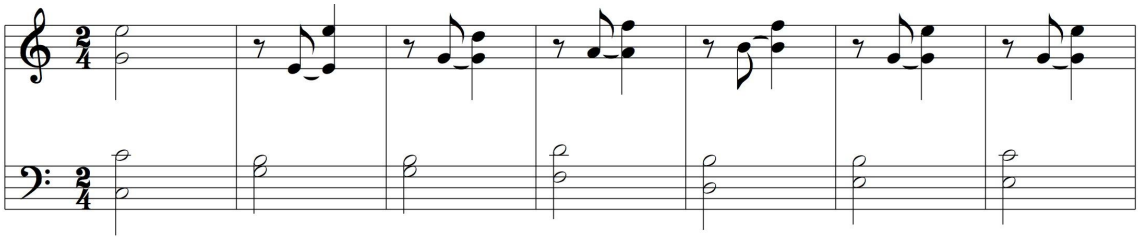

Figure 5.13: Representation of iteration using a correctly-implemented iterator.

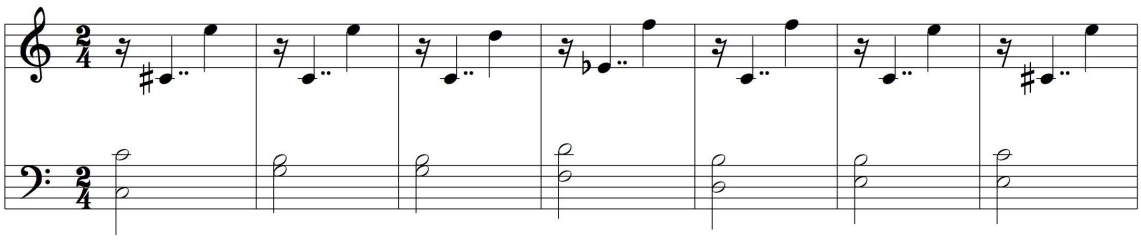

Figure 5.14: Representation of iteration using an incorrectly-implemented iterator.

5.4 Differences From Previous Auralizing Debuggers: Sonnet and CAITLIN

My system is fundamentally different from CAITLIN and Sonnet. CAITLIN is concerned with control flow and can only construct an auditory representation of a program by associating distinguishing sounds or musical signature to various language control constructs (selections and iterations). This means that other program features cannot be inspected aurally. Since my goal is to represent program state in sound, my target problem is different. CAITLIN does not allow a user to selectively decide which lines or sections of code should be auralized. This means that the entire program has to be auralized, and even short programs could take a long time to play back. In a real debugging situation, programmers would not monitor an entire program, but would choose candidate sections for close scrutiny.

Sonnet takes advantage of regular breakpoints, but depends on its visual programming language. Although that offers great flexibility to programmers who want to auralize a program, this requires a lot of work, even for auralization of very simple programs. Another reason that my sonified breakpoint tool differs from Sonnet is that Sonnet is based primarily on MIDI. Sonnet convert auralizations into MIDI data sent via an MIDI port to a suitable MIDI-equipped sound generator. Very few parameters for volume, pitch, and panning can be used. My sonified breakpoint tool, on the other hand, uses a synthesis tool which allows the programmer to arbitrarily add new sounds and parameters to the system. Moreover, using a synthesis tool provide more extensibility to my sonified breakpoint. For example, I can attach a special instrument to some sonified breakpoints, in order to silently collect global state information without playing sound.

5.5 Extension for External Music Tools

To provide support for other external rendering tools which are not built in the auralizing debugger, I have implemented two more kinds of breakpoints: *Update* breakpoints and *Point of Interest* (POI) breakpoints. Both send messages concerning the state of the program to the external rendering engines.

An Update breakpoint signifies a change in the state of some feature of the auralization. It has properties: a feature name, number of expressions and expressions that are evaluated to give the new values of the named feature. When an Update breakpoint is hit, the Eclipse debugger sends a message in the format

```
UPDATE Featurename NumberofExpressions
EvaluationofExpression1 EvaluationofExpression2...
```

to the external rendering engine, and waits for an acknowledgment. The acknowledgment will be either `RESUME`, indicating that the debugger should allow the program to continue until the next breakpoint is encountered, or `SUSPEND`, indicating that the debugger should suspend the program, update the debugger interface views, and await further interaction from the programmer, like an ordinary breakpoint.

A Point of Interest (POI) breakpoint indicates a point in the program's execution when some or all features of the auralization should be heard by the programmer. The term "Point of Interest" is taken from Vickers [63] , referring to one of the features of a Pascal programming construct to be auralized in CAITLIN. For example, the `IF` construct has four POIs:

- Entry to the `IF` construct.
- Evaluation of the conditional expression.
- Execution of selected statement.

- Exit from the `IF` construct.

For each construct type the first and last POIs always denote entry to and exit from the construct respectively. The difference between my POIs and Vickers's POIs is that my POIs focus on the program state not construct and my POIs are more flexible.

The properties of a POI breakpoint are the names of the features that should be auralized when the breakpoint is hit, the number of the features, the number of the expressions, and the expressions. When the POI breakpoint is hit, the debugger will send a message in the format

```
POI NumberofFeatures Featurename1 Featurename2...
NumberofExpressions EvaluationofExpression1  EvaluationofExpression2...
```

to the external rendering engine, and waits for an acknowledgment, as described for the Update breakpoint.

Update and POI breakpoints communicate runtime information on program state to an external engine capable of rendering that information in a format that goes beyond the capabilities of the Eclipse debugger. These external engines can perform auralizations that would be difficult to describe in the Structured Audio Orchestra Language. They can also feature other kinds of alternate representations, or a combination. Since they allow an external program to regulate the program execution in the debugger, they can even support the development of runtime analysis tools that enhance the capabilities of the debugger without invasive modifications.

The UML sequence diagram of how these two breakpoints works is provided in Figure 5.15.

5.5.1 Example of using this extension

I will give the musical representations of program state for detecting hidden error by using extension. This time, the `copySubtree` error described in the pilot study will be used as a hidden error. The Figure 5.16 shows the test program. The test program first builds a binary tree which is described in chapter 3 and then prints nodes of the created tree in preorder, and then makes a

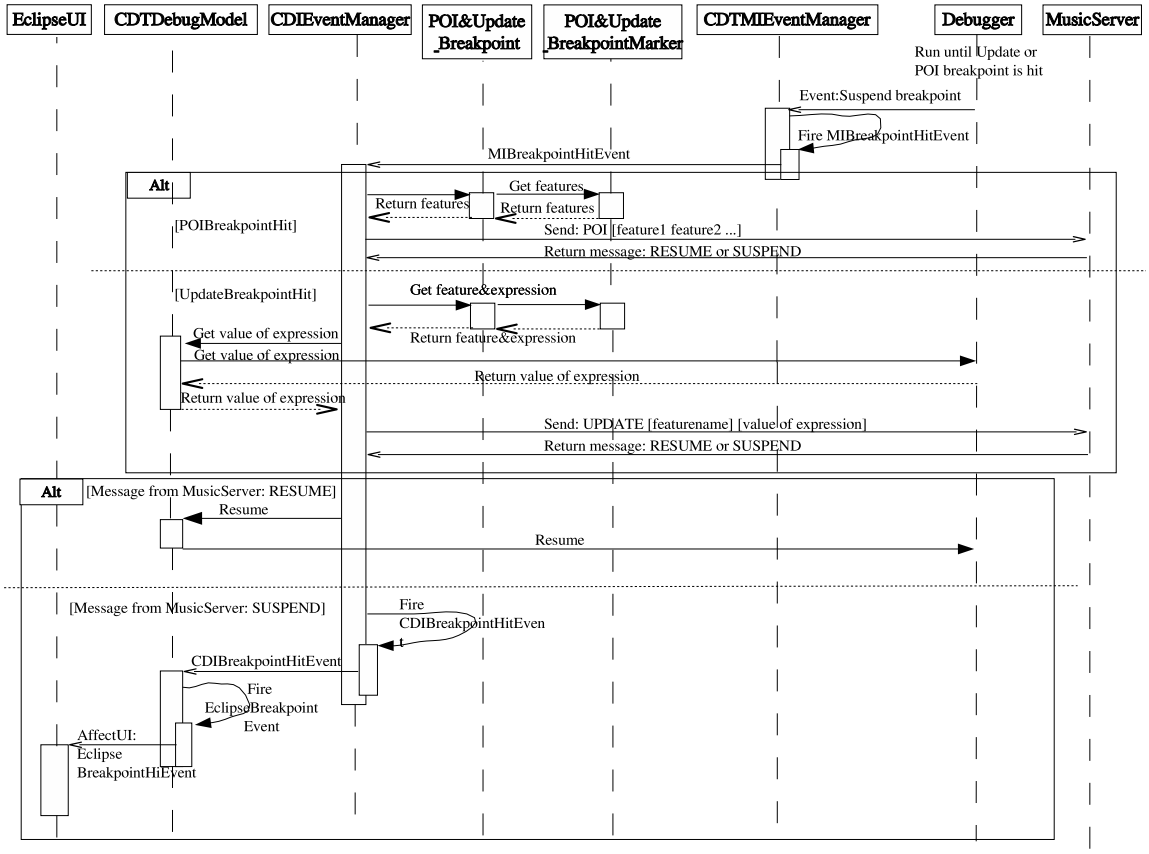

Figure 5.15: This graph shows how my debugger process events as they arrive asynchronously, when the update or POI breakpoint is hit.

copy of the tree, and then prints nodes of the copied tree in preorder. Both correct and incorrect implementations of `copySubtree` generate the same output shown in Figure 5.17.

Figure 5.18 shows a simple melody used to auralize a tree. Each node in the tree is represented by a single measure in the score, and the nodes are visited in a preorder traversal. The notes in each measure represent the left child, right child, and parent pointers for each node. A valid non-null pointer, a pointer to another node in the tree, is represented by an eighth note, as in Figure 5.19A, and a null pointer by a pair of staccato sixteenth nodes, as in Figure 5.19B. An invalid pointer, a non-null pointer that does not point to one of the nodes in the tree, is represented by an accented pair of simultaneous notes in the interval of a fifth (such as might be played as a double-stop on a violin), as in Figure 5.19C.

Music produced by the above auralization scheme can be used to represent the state of a complete copied tree. Here are two auralizations of two trees, the correct one in Figure 5.20, and the one with the bad parent pointers in Figure 5.21. These auralizations begin at measure 9 of the melody shown in Figure 5.18.

The similar auralization scheme can be used to track the progress of the `copySubtree` algorithm. Figure 5.22 shows how to set Update and POI breakpoints in the correct implementation of `copySubtree` algorithm. Figure 5.23 shows how to set Update and POI breaks in the incorrect implementation of `copySubtree` algorithm. Figure 5.24 and Figure 5.25 show messages received from external rendering tool for correct and incorrect implementations. Here are two auralizations of two trees, the correct one in Figure 5.26, and the one with the bad parent pointers in Figure 5.27.

5.6 Conclusion

State knowledge, which is often not immediately apparent in the text of the program, seems to be particularly difficult for novice programmers to develop, even in object-oriented programming languages. In this chapter, I have specified and described the motivation and design of a system for

```
typedef Tree<string> STree;
int main( void )
{
   // Better be able to declare instances of these types.
   STree::Position pos;

   // Make a STree and check its root.
   STree T( "moo" );
   pos = T.root();

   cout << "Building a binary tree" << endl;
   T.replaceElement( pos, "F1" );
   pos = T.addChild( T.root(), "C1" );
   T.addChild( pos, "A1" );
   T.addChild( pos, "B1" );

   pos = T.addChild( T.root(), "E1" );
   T.addChild( pos, "D1" );

   cout<<"Preorder print of the old tree"<<endl;
   preorderPrint( T, T.root() );

   cout << "Making copy" << endl;
   STree T2 = T;

   cout<<"Preorder print of the copied tree"<<endl;
   preorderPrint( T2, T2.root() );

   return 0;
} // end main
```

Figure 5.16: Test program for `copySubtree`. The test program first builds a binary tree which is described in chapter 3 and then print nodes of the created tree in preorder, and then make a copy of the tree, and then print nodes of copied tree in preorder. Both correct and incorrect implementations of `copySubtree` generate the same output.

```
Building a binary tree
Preorder print of the old tree
F1
 C1
  A1
  B1
 E1
  D1
Making copy
Preorder print of the copied tree
F1
 C1
  A1
  B1
 E1
  D1
```

Figure 5.17: Output of test program for `copySubtree`.

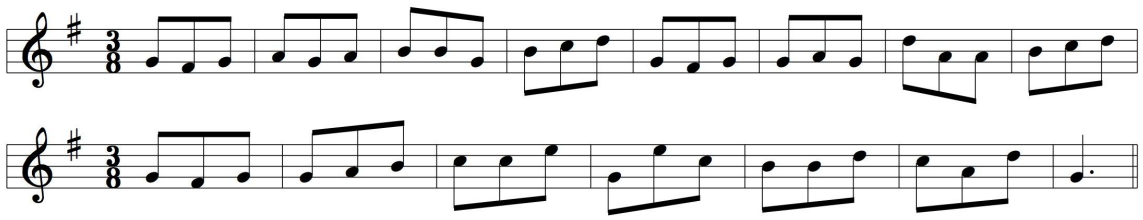

Figure 5.18: A simple melody used in the auralization of the tree.

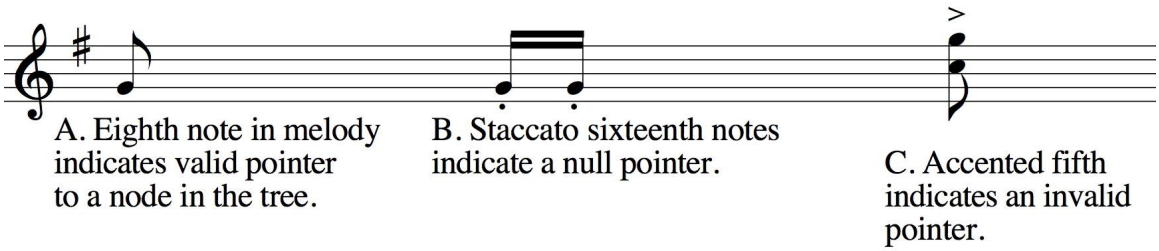

Figure 5.19: Musical signatures used in the auralization of the tree.

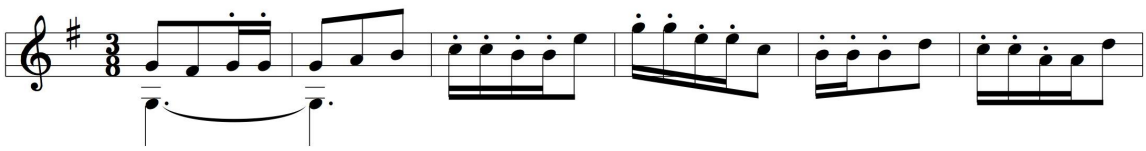

Figure 5.20: Representation of a correctly copied tree.

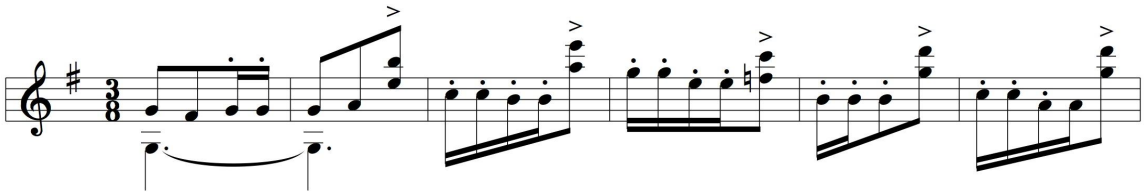

Figure 5.21: Representation of a corrupt, incorrectly copied tree.

the generation of sound representation of program state. I also explored the differences between previous program auralization tools and my tool, and the justification for external music control systems.

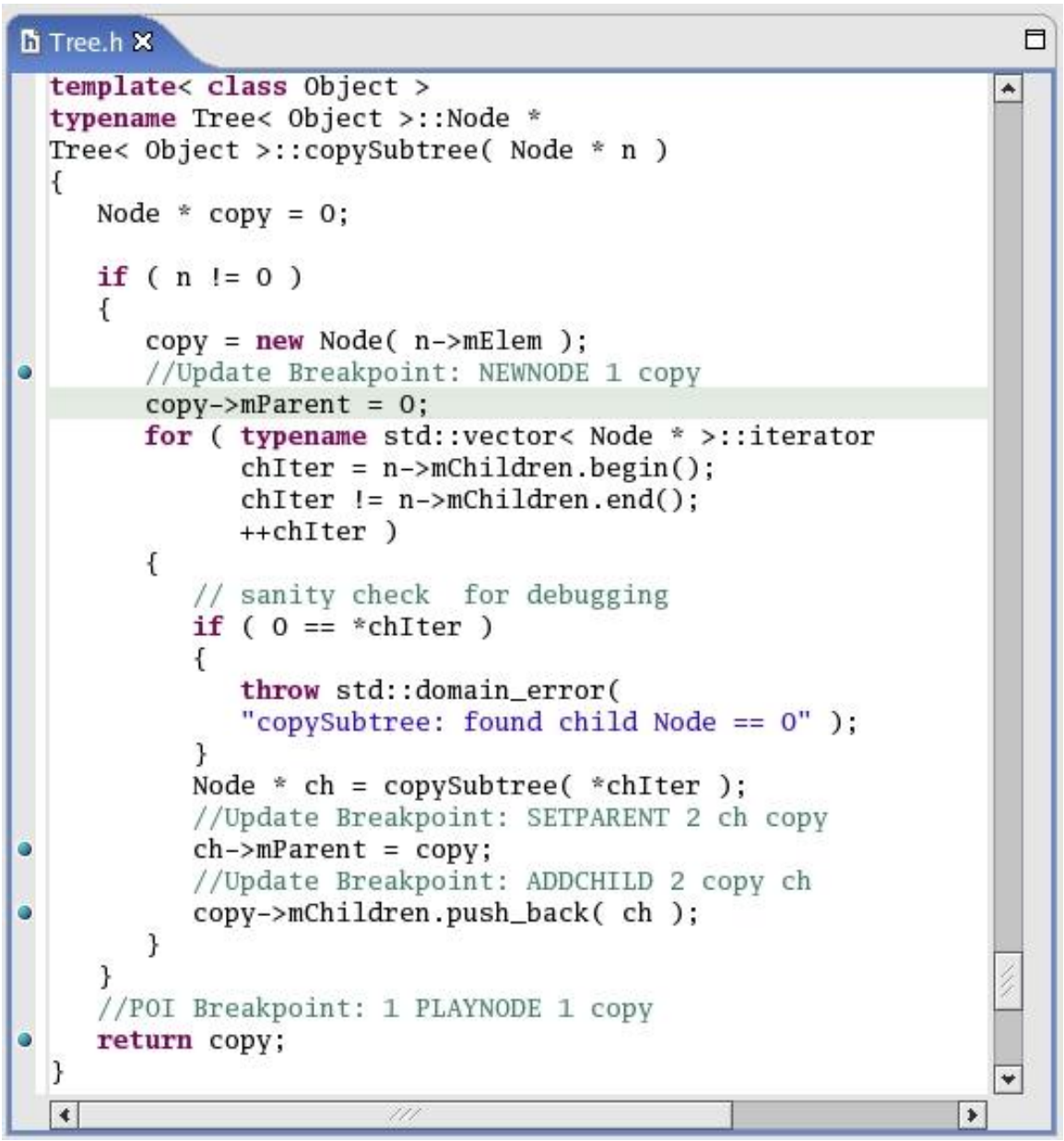

Figure 5.22: Setup of Update and POI breakpoints in the correct implementation of the `copySubtree` algorithm

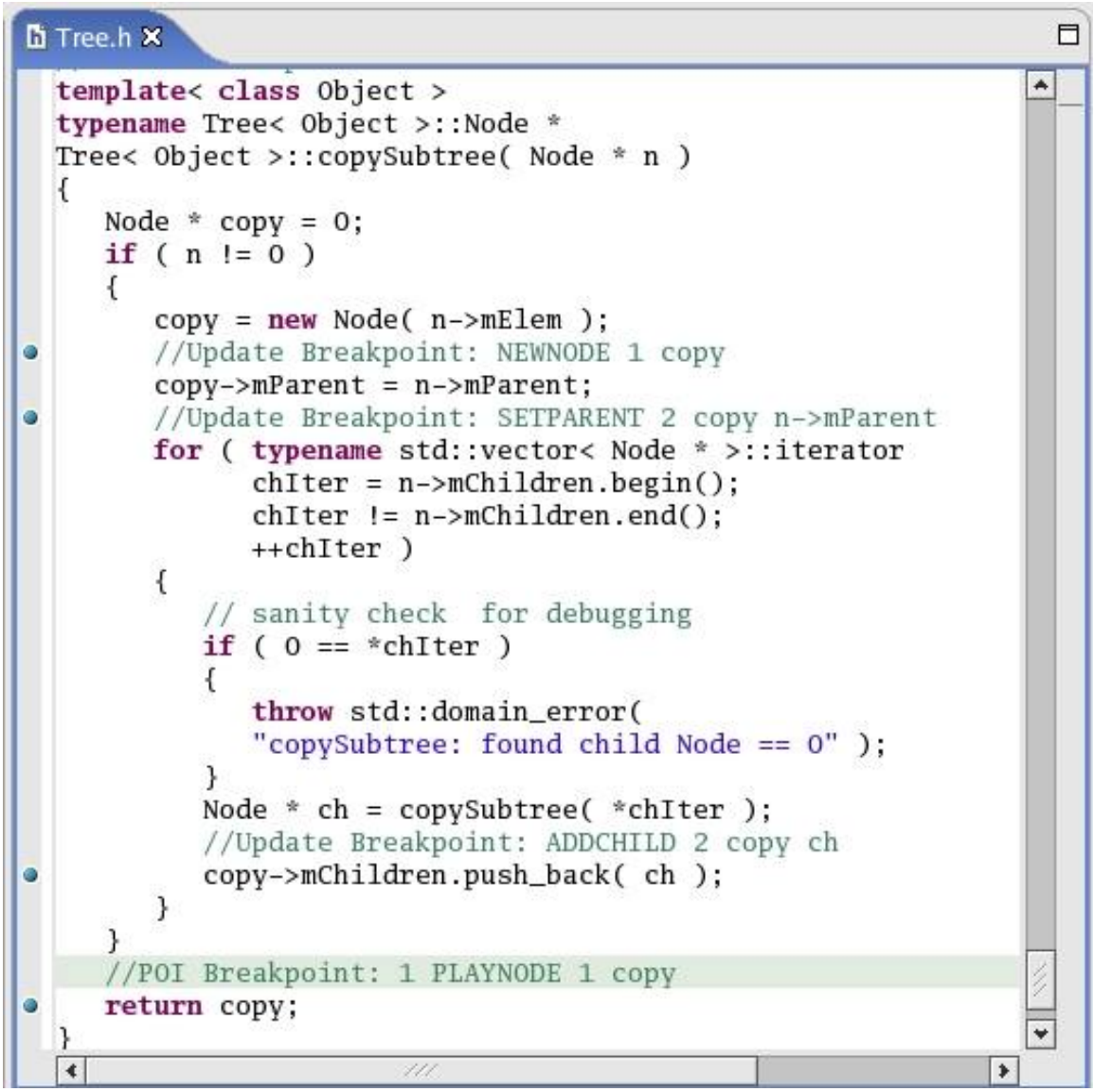

Figure 5.23: Setup of Update and POI breakpoints in the incorrect implementation of the `copySubtree` algorithm

```
UPDATE NEWNODE  0x8051640
UPDATE NEWNODE  0x8051658
UPDATE NEWNODE  0x8051670
POI 1 PLAYNODE 1  0x8051670
UPDATE SETPARENT  0x8051670 0x8051658
UPDATE ADDCHILD  0x8051658 0x8051670
UPDATE NEWNODE  0x8051688
POI 1 PLAYNODE 1  0x8051688
UPDATE SETPARENT  0x8051688 0x8051658
UPDATE ADDCHILD  0x8051658 0x8051688
POI 1 PLAYNODE 1  0x8051658
UPDATE SETPARENT  0x8051658 0x8051640
UPDATE ADDCHILD  0x8051640 0x8051658
UPDATE NEWNODE  0x80516a0
UPDATE NEWNODE  0x80516b8
POI 1 PLAYNODE 1  0x80516b8
UPDATE SETPARENT  0x80516b8 0x80516a0
UPDATE ADDCHILD  0x80516a0 0x80516b8
POI 1 PLAYNODE 1  0x80516a0
UPDATE SETPARENT  0x80516a0 0x8051640
UPDATE ADDCHILD  0x8051640 0x80516a0
POI 1 PLAYNODE 1 0x8051640
```

Figure 5.24: Messages received from the external rendering tool for correct implementation of `copySubtree`

```
UPDATE NEWNODE  0x8051640
UPDATE SETPARENT  0x8051640 0x0
UPDATE NEWNODE  0x8051658
UPDATE SETPARENT  0x8051658 0x80515b0
UPDATE NEWNODE  0x8051670
UPDATE SETPARENT  0x8051670 0x80515c8
POI 1 PLAYNODE 1  0x8051670
UPDATE ADDCHILD  0x8051658 0x8051670
UPDATE NEWNODE  0x8051688
UPDATE SETPARENT  0x8051688 0x80515c8
POI 1 PLAYNODE 1  0x8051688
UPDATE ADDCHILD  0x8051658 0x8051688
POI 1 PLAYNODE 1  0x8051658
UPDATE ADDCHILD  0x8051640 0x8051658
UPDATE NEWNODE  0x80516a0
UPDATE SETPARENT  0x80516a0 0x80515b0
UPDATE NEWNODE  0x80516b8
UPDATE SETPARENT  0x80516b8 0x8051610
POI 1 PLAYNODE 1  0x80516b8
UPDATE ADDCHILD  0x80516a0 0x80516b8
POI 1 PLAYNODE 1  0x80516a0
UPDATE ADDCHILD  0x8051640 0x80516a0
POI 1 PLAYNODE 1  0x8051640
```

Figure 5.25: Messages received from the external rendering tool for incorrect implementation of `copySubtree`

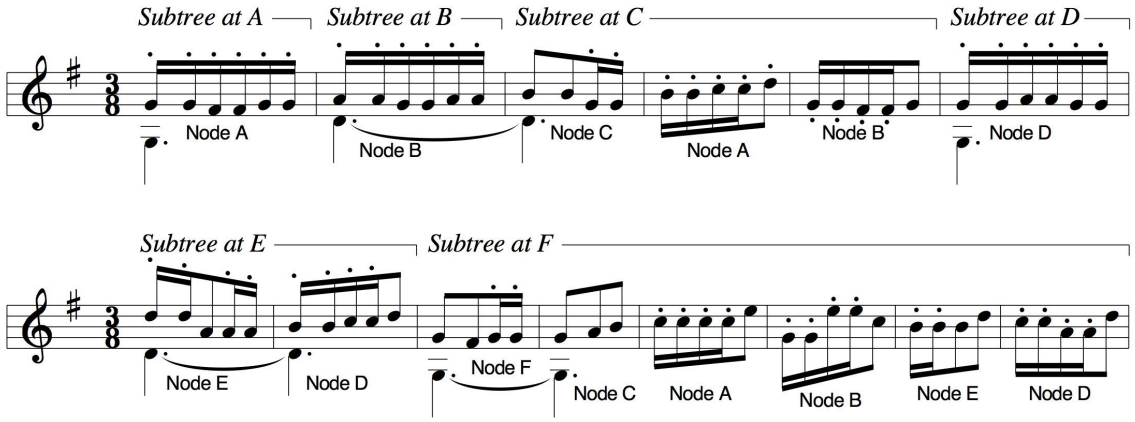

Figure 5.26: Representation of the execution of the correct implementation of `copySubtree` shown in Figure 3.8.

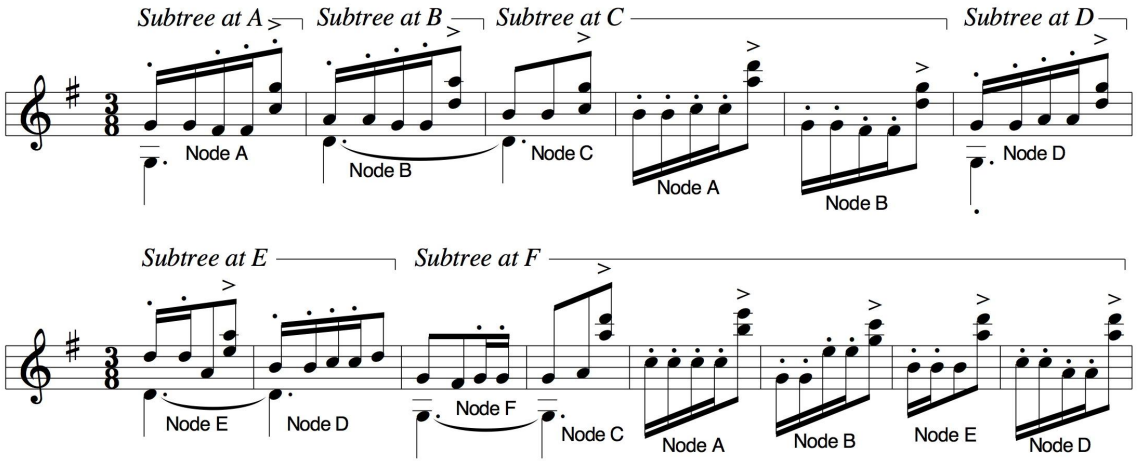

Figure 5.27: Representation of the execution of the incorrect implementation of `copySubtree` shown in Figure 3.9.

CHAPTER SIX

AN EXAMPLE EXPERIMENT

6.1 Introduction

I described my prototype auralizing debugger for novice programmers in Chapter 5. In this chapter, I propose an example experiment for determining the most effective way to represent different kinds of logic errors through sound. The lessons learned from this initial experiment will be used in the development of semantic music framework which describes what is the effective way to map program state to sound.

6.2 Research Questions and Hypotheses

6.2.1 Research Questions

There are two research questions as follows:

- Are people better able to associate a given auralization with certain types of logic errors ?
- Are people better able to associate certain types of auralizations with a given logic error ?

6.2.2 Hypotheses

The two hypotheses of the first question were formulated as follows:

- `A null hypothesis`: There is no difference on novice programmers'comprehensibility on auralizations which are based on the same scheme and targeted on different errors.
- `An alternative hypothesis`: There is difference on novice programmers'comprehensibility on auralizations which are based on the same scheme and targeted on different errors.

The Two hypotheses of the second question were formulated as follows:

- `A null hypothesis`: There is no difference on novice programmers'comprehensibility on auralizations which are based on different schemes and targeted on the same error.

	Error I	Error II
Auralization Scheme I	P1	P2
Auralization Scheme II	P3	P4

P denotes the average of identity score

Question one:
(1)
H0: P1=P2
H1: P1!=P2
(2)
H0: P3=P4
H1: P3!=P4

Question two:
(1)
H0: P1=P3
H1: P1!=P3
(2)
H0: P2=P4
H1: P2!=P4

Figure 6.1: Hypothesis in mathematical way.

- `An alternative hypothesis`: There is difference on novice programmers'comprehensibility on auralizations which are based on different schemes and targeted on the same error.

Figure 6.1 shows the mathematical way of addressing the hypotheses.

6.3 Research Design

6.3.1 Experimental Principles

One of the main advantages of an experiment is the control of participants, tasks, and instrumentation. This ensures that I am able to draw more general conclusions. Other advantages include the ability to perform statistical analysis using hypothesis testing methods and opportunities for replication. To ensure that I make use of these advantages, I developed a process to support me in

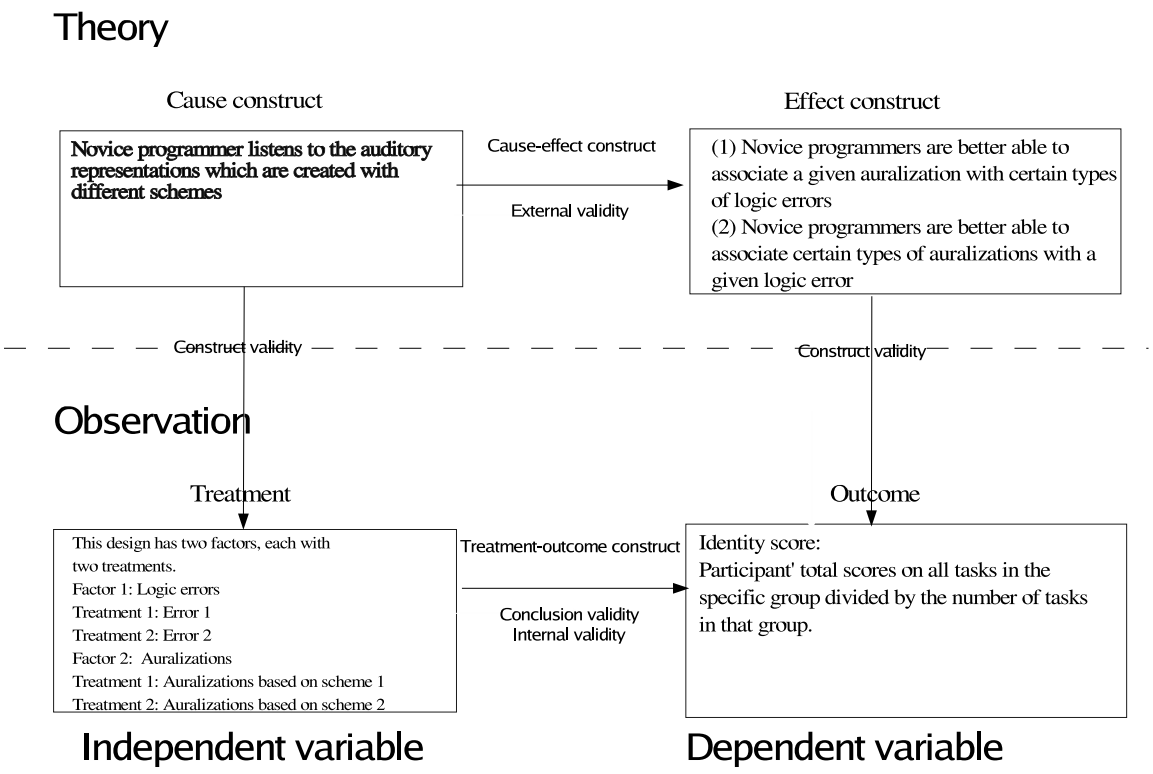

Figure 6.2: Experiment Principles

these objectives. The basic principles behind my experiment are illustrated in Figure 6.2.

6.3.2 Independent and Dependent Variables

Before conducting the experiment, I choose the dependent and independent variables described in Figure 6.3.

6.3.3 Experimental Design

I designed an experiment based on two factors, each with two treatments (called a 2 by 2 factorial design. I will randomly assign participants to each combination of the treatments (see Figure 6.1). The first factor is the logic error, and its treatments are Error I and Error II. The second factor is the auralizations, and its treatments are auralizations based on Scheme I and Scheme II.

	Name	Scale Type	Notes
Independent Variables	Auralizations	Nominal	Treatment 1: Auralizations based on scheme 1 Treatment 2: Auralizations based on scheme 2
	Errors	Nominal	Treatment 1: Error 1 Treatment 2: Error 2
	Sound Sensitivity score	Ordinal	Levels of measurements: Sound sensitivity score represents the score attained by subjects on a musical knowledge test. One mark was given for each correct answer resulting in a range of 0-15
Dependent Variables	Identity Score	Ratio	If subject correctly answer the question, the subject will get 1 point; otherwise subject will get 0 point. The is called raw score. The identity score is equal to a subject's total raw scores on all objects divided by the number of objects.

Figure 6.3: Choice of independent and dependent variables

6.3.4 Validity Threats

Threats to the conclusion validity of this experiment include low statistical power and the reliability of measures due to small sample size, since we will probably recruit less than 40 students. Measures on participants' ability to recognize sound patterns are very subjective.

Threats to the internal validity are influences that can affect the independent variable with respect to causality, without the researcher's knowledge. Because the participants in my experiment are volunteers, they are probably more motivated and suited for a new task than most novice programmers. Hence, the selected group is not representative of novice programmers. The selected auralizations may not be the representative of all the auralizations. Moreover, the participants may not be trained enough.

Construct validity refers to the process of generalizing the result of the experiment to the concept or theory behind the experiment. In my experiment, measures on the participants' ability to recognize sound patterns are very subjective, so there is a risk of measurement bias.

Threats to external validity are conditions that limit my ability to generalize the results of my experiment to the field of education. Since convenience sampling is used to choose volunteers from an introductory programming, participants may not be the representative of the whole population of novice programmers.

6.4 Participants

participants who will take part in this experiment are all volunteers. They will be students at the Department of Electrical Engineering and Computer Science at Washington State University, enrolled in introductory computer programming courses.

6.5 Material and Tasks

In the pilot study, I found that data structure implementations like the tree problem posed problems for all of the participants. Some spent a lot of time debugging, not always successfully. Others

```
////////////////////////////////////////////////////////////////////
/// end
/// Returns an iterator positioned just beyond the end of the container
/// \pre    Vecotr exists.
/// \post   Position is returned.
/// \param  None.
/// \return An iterator positionad just beyond the end of the container.
////////////////////////////////////////////////////////////////////
template <class T> typename Vec<T>::iterator Vec<T>::end()
{
   VecIterator<T> iter( *this, avail );
   return iter;
}
```

Figure 6.4: Correct implementation of `vector.end()`

never discovered the errors. In this experiment, I will present auralization examples of vector and iterator validation.

Instead of source programs, I will give an introduction on the functionality of each program and its auditory representations to the participants. For each auralization, I will ask participants to answer a question with multiple possible answers. The participants receive a point for a correct answer; otherwise, the participant receives no points. The auralizations will be designed with a maximum length of approximately one minute.

The error types in vector and iterator validation are as follows:

- `vector.end()` returns default iterator. Figure 6.4 and Figure 6.5 show two implementaions, one correct and one incorrect, of the function `vector.end()`.
- Increment operator does not increment, so the iterator never advances. Figure 6.6 and Figure 6.7 show two implementations, one correct and one incorrect, of the increment operator of iterator.

The music schemes are as follows:

```
template <class T> typename Vec<T>::iterator Vec<T>::end()
{
   //returns default iterator.
   VecIterator<T> iter;
   return iter;
}
```

Figure 6.5: Incorrect implementation of `vector.end()`

```
////////////////////////////////////////////////////////////////////
/// Pre-increment operator
/// \pre    interator exists
/// \post   Pointer advanced by one memory location
/// \param  Nothing
/// \return reference to itself
////////////////////////////////////////////////////////////////////
template <class T> VecIterator<T>& VecIterator<T>::operator++( void )
{
    ++mPtr;
    return *this;
}
```

Figure 6.6: Correct implementation of increment operator of iterator

```
template <class T> VecIterator<T>& VecIterator<T>::operator++( void )
{
    //Increment operator does not increment,
    //so the iterator never advances.
    return *this;
}
```

Figure 6.7: Incorrect implementation of increment operator of iterator

- The first scheme, called `Voicing`, changes the way the chords are voiced according to the state of the iterator. Voicing use the same musical signatures to auralize iterator state shown in Figure 5.11. Figure 6.9 shows the chord sequence for the auralization of iterator in the scheme of `Voicing`.

- The second scheme, called `Orch`, changes the way that other instruments accompany the piano according to the state of the iterator. The piano always plays the same thing, shown in the Figure 6.10. The piano part does not vary with the program state. Only the flute and trombone parts vary with the program state. Access to the first element is marked by a chord played by all three instruments, the flute and trombone adding long notes that last the whole measure, like the piano left hand. Access to other valid elements is marked by the flute doubling (playing the same notes as) the piano right hand, one octave higher. The trombone plays nothing. Access to one-past the last element of the vector (the first incorrect access) is marked by a dissonant chord: the flute and trombone notes make a chord that is dissonant with respect to the piano, and their notes come in one beat later than the piano left hand. Access to any other out-of-range position is marked by the trombone playing an eighth-note figure that contrasts harmonically and melodically with the piano right hand. The flute plays nothing.

Figure 6.8 shows the test program for both `vector.end()` and increment operator of iterator. Figure 6.11 shows the music played by the correct implementations in `Voicing`. Figure 6.12 shows the music played by the correct implementations in `Orch`.

If the increment operator of the iterator doesn't increment itself, the iterator always refers to the first element in the vector. Figure 6.13 and Figure 6.14 show the two music representations played when the increment operator doesn't increment in the schemes of `Voicing` and `Orch`. If `vector.end()` returns a default iterator, the iterator moves from the first valid position in the

```
void printRange(Vec<int>::iterator pos, Vec<int>::iterator end)
{
    cout << "Printing objects in the vector"<< endl;
    while(pos != end)
    {
       cout<<*pos<<endl;
       ++pos;
    }
}
int main(void)
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   Vec< int > vi(100);
   vi.clear();
   for ( int k = 0; k < SIZE; k++ )
   {
      vi.push_back( a[k] );
   }
   printRange(vi.begin(),vi.end());
   return 0;
}
```

Figure 6.8: Test program for both `vector.end()` and increment operator of iterator.

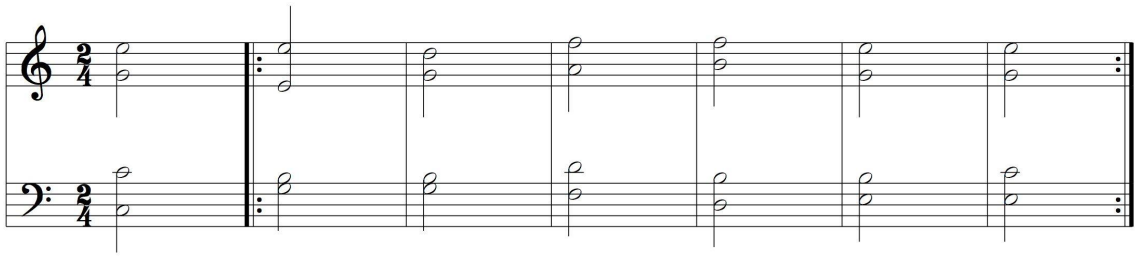

Figure 6.9: Chord sequence for the auralization of iterator in the scheme of `Voicing`.

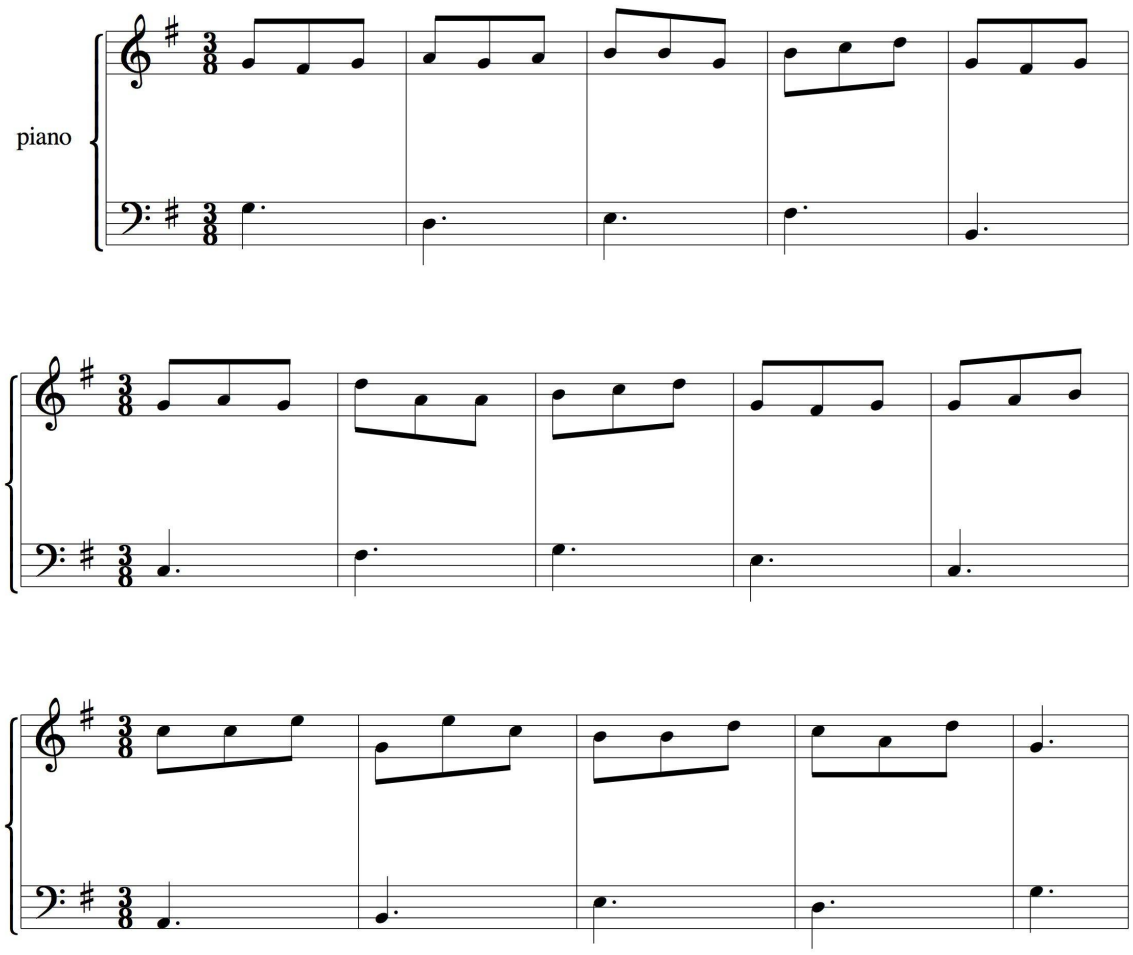

Figure 6.10: The piano in `Orch` always plays the same thing.

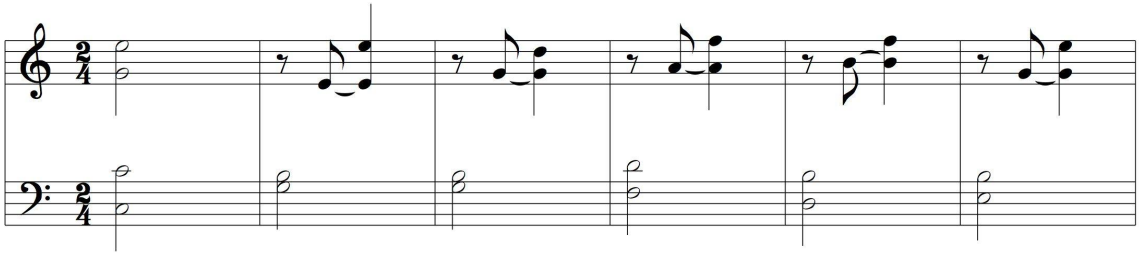

Figure 6.11: The music played by the correct implementations in `Voicing`.

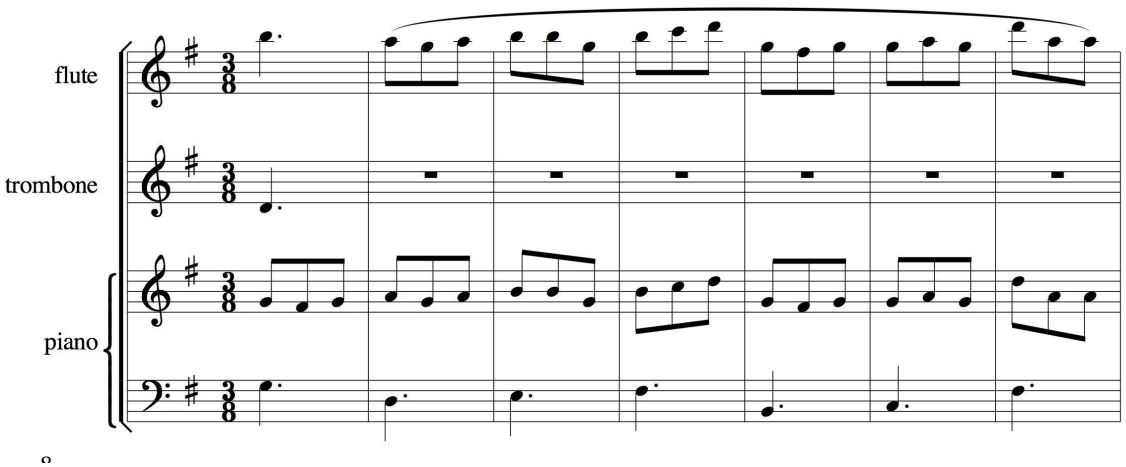

Figure 6.12: The music played by the correct implementations in `Orch`.

vector to last valid position in the vector, and then it does not stop and keeps moving until segmentation fault happens. Figure 6.15 and Figure 6.16 show the music played when `vector.end()` returns a default iterator in the schemes of `Voicing` and `Orch`.

6.6 Procedure

The experiment comprises four sessions:

- Introduction and tutorial
- Pretest on ability of sound pattern recognition and participant experience questionnaire.
- Listening tests
- Participant feedback

At the start of the experiment, each participant is given a workbook that contains:

- The full written text of the introduction and tutorial session.
- Pretest on ability of sound pattern recognition and participant experience questionnaire.

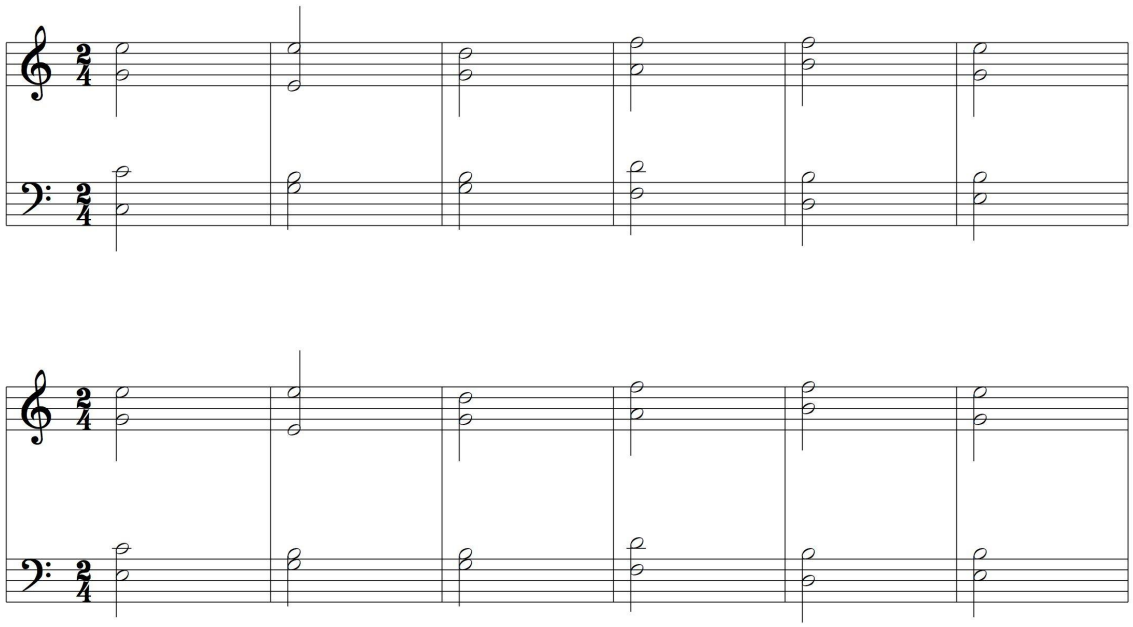

Figure 6.13: The music played when the increment operator doesn't increment in the scheme of `Voicing`

- Problem descriptions for the listening exercises, comprising a textual description of corresponding data structures, the auralization scheme, and questions.
- A space for detailed participant responses and general feedback on their experience of the experiment.

6.6.1 Introduction and Tutorial

The participants have no prior experience in program auralization, so a briefing session will be given. The tutorial explains the philosophy of the technique of program auralization and gave examples of the various auralizations used. An explanation of how the auralizations are constructed and how they represent the program state will also be given.

6.6.2 Pretest and Questionnaire

In this session, participants first complete the participant data questionnaire, which is used to gather information about their programming experience. Later, a pretest of the participants' ability to

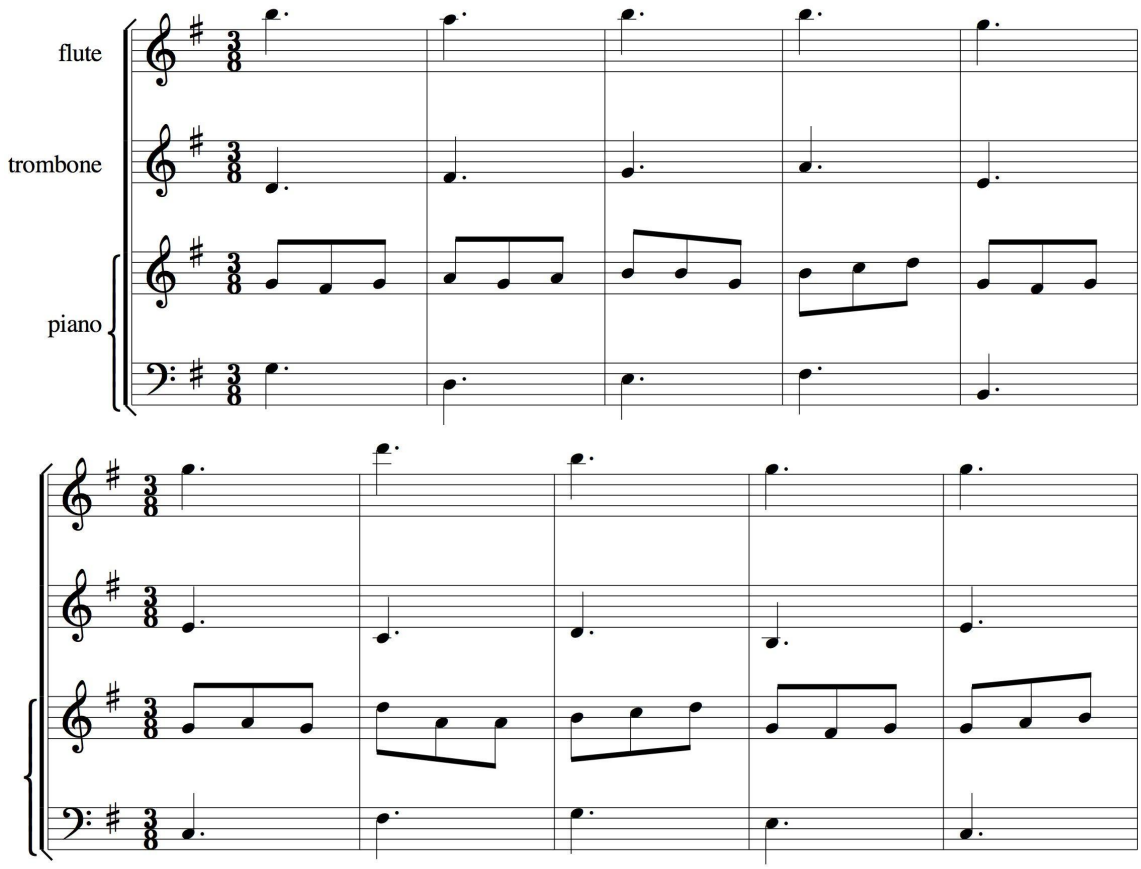

Figure 6.14: The music played when the increment operator doesn't increment in the scheme of `Orch`

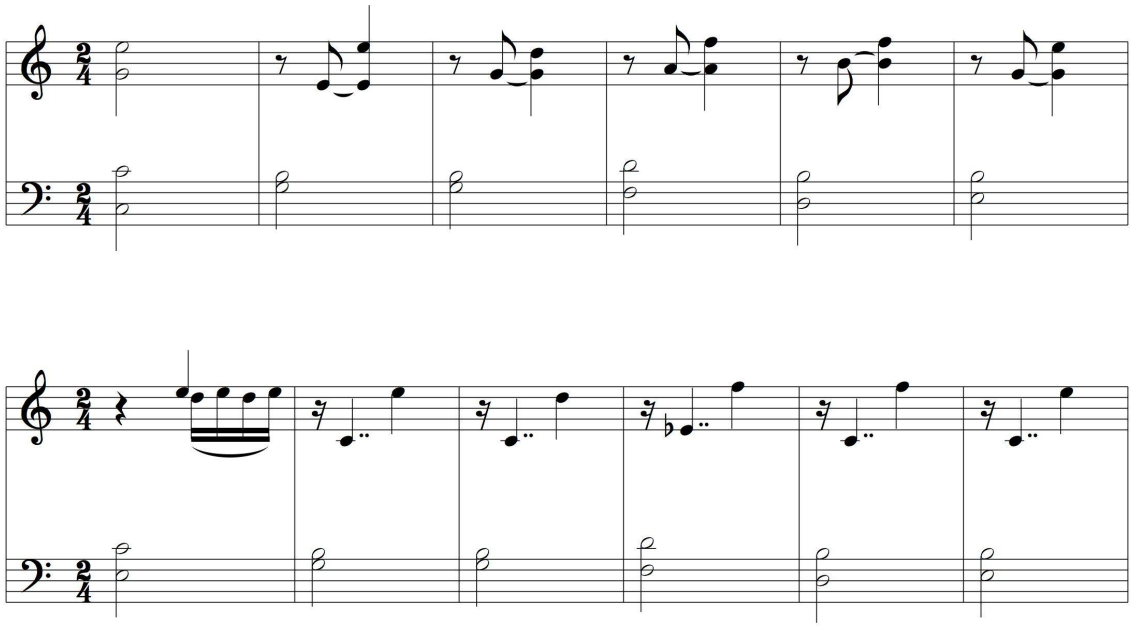

Figure 6.15: The music played when `vector.end()` returns a default iterator in the scheme of `Voicing`

recognize sound patterns will be conducted to find any differences.

6.6.3 Listening Tests

The auralizations will be prepared prior to the experiment and stored on the web application as an audio file. Participants download them based on the order of tasks performed that are given to the specific participant. Windows Media Player will be installed on all machines to provide participants facilities for playing, stopping, pausing, and rewinding the auralizations.

6.6.4 Feedback

Following the listening tests, participants are invited to write down their responses to the experiment on a page in the workbook. Finally, participants hand in their workbooks. The participants' responses are later analyzed to see how they interpreted the auralizations.

6.6.5 Data Collection

All the data will be collected manually by the experimenter.

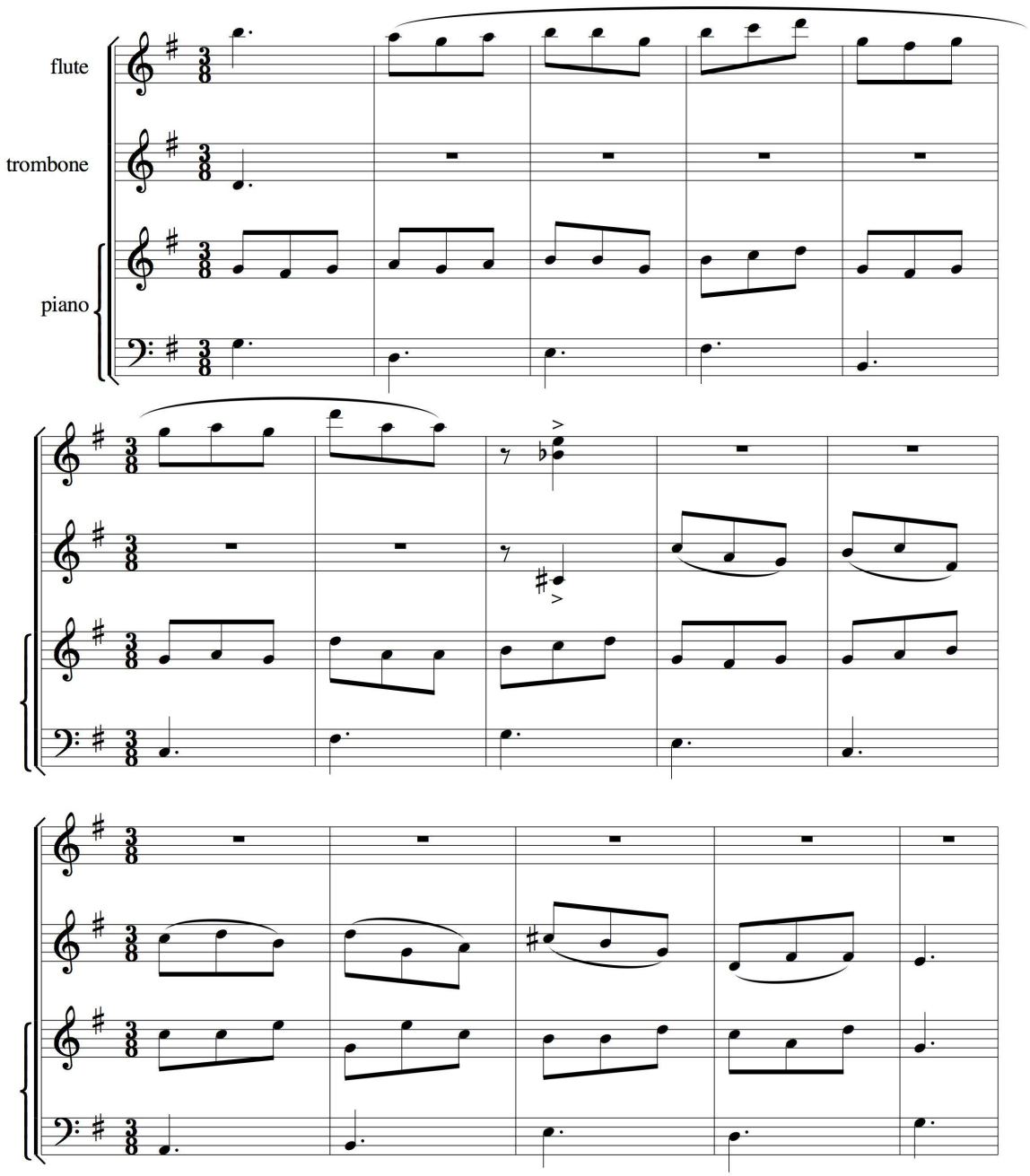

Figure 6.16: The music played when `vector.end()` returns a default iterator in the scheme of `Orch`

6.7 Analysis

6.7.1 Descriptive Statistics and Data Set Reduction

Descriptive statistics and box plots on the data will be used to describe and graphically present how concentrated or spread out the data set is, and assist me to better understand the nature of the data and to identify abnormal or false data points(called outliers).

6.7.2 Hypothesis Testing

To test my hypotheses, I will first check to see if the data is normally distributed. The normal probability plot will be used to illustrate this, showing whether the data is skewed, with shorter or longer tails than expected. The correlation coefficient of the points on the normal probability plot will be compared to a table of critical values, providing a formal test of the hypothesis that the data come from a normal distribution.

If the data are approximately normally distributed, an ANOVA (analysis of variance) on the mean of identity score will be used. If not, the procedures of the two-way layout in nonparametris statistics will be used.

6.7.3 Post-Analysis if Necessary

I anticipate that other factors such as participants' ability of sound pattern recognition, the experience of the participants and the complexity of the auralization will not affect their ability to make use of program auralization. But if I find that participants merely guessed the identities, I need to carry out the post-analysis on those factors to see what factors affect the identity score.

6.8 Result

This experiment will determine the effect of auralizations based on different schemes, on the different logic errors, and how participant interpreted the auralizations. The information abtained in this study will guide us in the development of a semantic music framework for describing how program state can be effectively mapped with sound.

CHAPTER SEVEN

CONCLUSION AND FUTURE WORK

The major strength of this work over prior program auralization systems is that it is based on empirical results. The final goal of this project seeks to address the question of whether sound can be used as a communication medium to assist novice programmers in introductory computer programming classes to improve debugging performance. To answer this question, I first carried out a pilot study on novice programmer debugging behavior and common logic errors. I found that while students may understand that a program does not work, and may even know which general part of the program is at fault, they still have insufficient understanding of how the program executes and fail to develop a good mental model of the evolving state of a program, especially when advanced data structures are involved. The findings from the pilot study guided the development of a prototype auralizing debugger. The proposed experiment will show whether this kind of auralization can be understandable to novice programmers.

The development of this auralizing debugger is a major contribution to the field. Prior to this research, there was no effort to construct a programming and debugging environment in which the user has full control over the application of visualization and auralization techniques. This equal-opportunity interface will offer a variety of communication media from which the user can select an appropriate mix to match their capabilities and limitations. The extension for external rendering tools can allow researchers to easily experiment with different melodies and auralization schemes. Moreover, since the extension allows an external program to regulate program execution in the debugger, it could even support the development of runtime analysis tools that enhance debugger capabilities without extensive modifications.

It is clear that there is a need for much more experimentation in the future. The best way to make my audio enhanced debugger truly useful is for me to continue my research on novice programmers' behavior at WSU. This data will represent a substantial body of work on how students

learn to develop and debug programs in C++ object-oriented language. New tools designed to address specific issues in the pilot study findings will be integrated into the experimental version of my audio enhanced debugger to help students to identify common programming errors more quickly, discourage unproductive debugging activities such as staring at code, and encourage more productive ones. I plan to run empirical studies on enhanced debugging tools in order to evaluate their effectiveness in reducing time spent finding and fixing common errors and bringing a program to a complete and correct state. This cycle of pilot studies, tool deployment and empirical evaluation will provide a consistent test bed for program auralization research, and will provide instructors with data used to improve instruction in program debugging.

Musical motifs that can communicate information about program execution and program state should be developed. Longitudinal studies should be carried out in introductory computer programming courses to help me discover how students' experience with auditory display techniques affects their ability to program and debug. I also plan to investigate how music, other non-speech audio, and graphical display techniques can be combined.

Auditory display is a field of computer science which is currently so poorly understood that it may take years before researchers are capable of fully exploiting this underutilized medium. It is my hope that the creation of my audio enhanced development and debugging environment will help novice programmers understand program state with program state auralization, and allow researchers to explore new possibilities and directions.

BIBLIOGRAPHY

[1] James L. Alty. Can we use music in computer-human communication? In *Proceedings of the HCI'95 Conference on People and Computers X*, pages 409–423, Huddersfield, United Kingdom, April 1996.

[2] Alty J.L., Rigas D., and Vickers P. Using music as a communication medium. In *Proc. CHI 97: Conference on Human Factors in Computing Systems*, pages 22–27, Atlanta, March 1997.

[3] Alty J.L. and Vickers P. Using music to communicate computing information. *Interacting with Computers*, 14(5):434–455, 2002.

[4] Walker B.N., Kramer G., and Lane D.M. Psychophysical scaling of sonification mappings. In *Proceedings of the ICAD 2000 Sixth International Conference on Auditory Display. International Community for Auditory Display*, pages 2–5, Atlanta, GA, April 2000.

[5] Bock D.S. *Auditory software fault diagnosis using a sound domain specification language*. PhD thesis, Syracuse University, 1995.

[6] Bouvier P. Visual tools to debug prolog iv programs. *Analysis and visualization tools for constraint programming: constraint debugging*, pages 177–190, 2000.

[7] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualization for novice pascal programmers. In *IEEE Workshop on Visual Languages*, pages 192–198, October 1988.

[8] Brown M.H.and Hershberger J. Color and sound in algorithm animation. *Computer*, 25(12):52–63, 1992.

[9] Cameron M., Garca de la Banda, K. Marriott, and P. Moulder. Vimer: a visual debugger for mercury. *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, August 2003.

[10] Carro, M. and Hermenegildo M. Apt tool. *Analysis and visualization tools for constraint programming: constraint debugging*, pages 237–252, 2000.

[11] Carver and Klahr. Children's acquisition of debugging skills in a logo environment. Journal of educational computing research.

[12] Ryan Chmiel and Michael C. Loui. Debugging: from novice to expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 17–21. ACM Press, 2004.

[13] Claudius M. Kessler and John R. Anderson. Model of novice debugging in lisp. In *Empirical Studies of Programmers*, pages 198–212, 1986.

[14] Cohen J. Monitoring background activities. In Kramer, editor, *Auditory Display*, volume XVIII, pages 499–532. Addison-Wesley, 1994.

[15] Corritore C.L. and Wiedenbeck S. What do novices learn during program comprehension. *International Journal of Human-computer Interaction*, 3(2):199–222, 1991.

[16] Deutsch D. and Feroe J. The internal representation of pitch sequences in tonal music. In *Psychological Review*, volume 88, pages 503–532, 1981.

[17] Boardman D.B. and Mathur A.P. Preliminary report on design rationale, syntax, and semantics of lsl: A specification language for program auralization. Technical report, Dept. of Computer Science, Purdue University, Sept. 1993.

[18] Dibben N. The cognitive reality of hierarchic structure in tonal and atonal music. *Music Perception*, 12(1):1–26, 1994.

[19] Digiano C.J. and Baecker R.M. Program auralization: sound enhancements to the programming enviroment. In *Proc. Graphics Interface'92*, pages 44–52, 1992.

[20] Dimitrios I. Rigas and James L. Alty. The use of music in a graphical interface for the visually impaired. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 228–235, 1997.

[21] Marc Eisenstadt. Tales of debugging from the front lines. In *Empirical Studies of Programmers*, December 1993.

[22] Eisenstadt M. and Rrayshaw M. The transparent prolog machine(tpm): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, December 1998.

[23] Lerdahl F. and Jackendoff R. *A Generative Theory Of Tonal Music*. MIT Press, Cambridge, MA, 1983.

[24] Francioni J., Albright L., and Jackson J. Debugging parallel programs using sound. *SIGPLAN Notices*, 26(12):68–75, December 1991.

[25] Michael T. Goodrich, Roberto Tamassia, and David M. Mount. *Data Structures and Algorithms in C++*. Addison Wesley, 2004.

[26] Gould J.D. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7:151–182, 1975.

[27] Gould J.D. Novices' debugging when programming in pascal. *International Journal of Man-Machine Studies*, 33:707–724, 1990.

[28] Wayne C. Gramlich. Debugging methodology: session summary. In *Proceedings of the Symposium on High-Level Debugging*, June 1983.

[29] L. Gugerty and G. Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 171–174. ACM Press, 1986.

[30] Isoda S., Shimomura T., and Ono Y. Vips: A visual debugger. *IEEE Software*, 4:8–15, 1987.

[31] J. A. Jackson and J. M. Francioni. Synchronization of visual and aural parallel program performance data, in audiotory display. In Gregory Kramer, editor, *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Addison-Wesley, 1994.

[32] J.A. Jackson and J.M. Francioni. Aural signatures of parallel programs. In *Twenty-Fifth Hawaii International Conference on System Sciences*, pages 218–229, 1992.

[33] David H. Jameson. Sonnet: Audio-enhanced monitoring and debugging. In Gregory Kramer, editor, *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Addison-Wesley, 1994.

[34] R. Jeffries. A comparison of the debugging behavior of expert and novice programmers. A paper presented at the AERA Annual Meetting.

[35] Jeffroes R. Computer program debugging by experts, 1981. Paper presented at the Annual Meetings of the Psychosomatic Society.

[36] Jeffroes R. A comparison of the debugging behavior of expert and novice programmers, 1982. Paper presented at the meetings of the American Educational Research Association.

[37] Andrew J.Ko and Brad A. Myers. Development and evaluation of a model of programming errors. In *Human Centric Computing Languages and Environments. Proceedings, IEEE Symposium*, pages 7–14, 2003.

[38] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: programming as a multimedia experience*, chapter 19, pages 277–293. The MIT Press, 1999.

[39] Francioni J .M. and Rover D. T. Visual-aural representations of performance for a scalable application program. In *Proc. High Performance Computing Conference*, pages 433–440, 1992.

[40] Nanja M. and Cook C.R. An analysis of the on-line debugging process. In *Empirical studies of programmers: second workshop*, pages 172–183, 1987.

[41] Tara M. Madhyastha and Daniel A. Reed. Data sonification: Do you see what I hear. *IEEE Software*, 12(2), March 1995.

[42] Thomas G. Moher. Provide: a process visualization and debugging environment. *IEEE Transaction on Software Engineering*, 14(6):849–857, June 1988.

[43] Mostrm Jan-Erik and David A. Carr. Programming paradigms and program comprehension by novices. In *PPIG '98 Workshop*, 1997.

[44] Brad A. Myers, John F. Pane, and Andy KO. Natural programming languages and environments. *Communications of the ACM*, 47(9), 2004.

[45] Pennington N. Comprehension strategies in programming. In G.M.Olson, S.Sheppared, and E.Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, Norwood,Nj, Ablex, 1987.

[46] Pennington N. Stimulus structures and mental representations in expert comprehension of computer programs. In *Cognitive Psychology*, number 19, pages 295–341, 1987.

[47] Nanja M. and Cook C.R. An analysis of the on-line debugging process. In *Empirical Studies of Programmers: Second Workshop*, pages 172–184, 1987.

[48] Narmour E. Some major theoretical problems concerning the concept of hierachy in the analysis of tonal music. In *Music Perception*, volume 1 of *2*, pages 129–199, 1984.

[49] Parncutt R. *Harmony: A Psycho-Acoustical Approach*. Springer-Verlag, Berlin, 1989.

[50] Pazel D.P. Ds-viewer: an interactive graphical data structure presentation facility. *IBM Systems Journal*, 28:307–323, 1989.

[51] Nancy Pennington and Beatrice Grabowski. The tasks of programming. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 45–62. Academic Press, London, 1990.

[52] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the Seventh Workshop on Empirical Studies of Programmers*, pages 124–139. ACM Press, 1997.

[53] Steven P. Reiss. Pecan: Program development systems that support multiple views. *Pro. 7th ICSE*, pages 324–333, 1984.

[54] Eric D. Scheirer and Barry L. Vercoe. SAOL: The MPEG-4 structured audio orchestra language. *Computer Music Journal*, 23(2):31–51, 1999.

[55] C. Oz Schulte. Explorer: a visual constraint programming tool. In *Proceedings of the 14th International Conference on Logic Programming*, pages 286–300. MIT press, 1997.

[56] D. H. Sonnenwald, B. Gopinath, G. O. Haberman, W. M. Keese, and J. S. Myers. InfoSound: An audio aid to program comprehension. In *Proc. Twenty-Third Hawaii International Conference on System Sciences*, pages 541 – 546. IEEE Computer Society Press, 1990.

[57] Sougata Mukherjea and John T.Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1(3):215–244, 1994.

[58] John Stasko, John Domingue, Marc H. brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*, chapter 1, pages 1–43. The MIT Press, 1999.

[59] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*, chapter 18, pages 259–277. The MIT Press, 1999.

[60] John Stasko, John Domingue, Marc H. brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*, chapter 10, pages 137–143. The MIT Press, 1999.

[61] Swain J.P. The need for limits in hierarchical theories of music. In *Music Perception*, volume 4, pages 121–148, 1986.

[62] Vessey I. Expertise in debugging computer program: a process analysis. *International Journal of Man-Machine Studies*, 23:459–494, 1990.

[63] P. Vickers and J. L. Alty. Caitlin: A musical program auralisation tool to assist novice programmers with debugging. In *Proc. ICAD*, 1996.

[64] P. Vickers and J. L. Alty. Musical program auralisation: Empirical studies. In *Proc. ICAD*, 2000.

[65] Paul Vickers and James L. Alty. Siren songs and swan songs: debugging with music. *Commun. ACM*, 46(7):86–93, 2003.

[66] Jessica R. Weinstein and Perry R. Cook. Faust: A framework for algorithm understanding and sonification testing. In *The Fourth International Conference on Auditory Display*, Palo Alto, California, November 1997.

[67] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[68] Andreas Zeller and Dorothea Lutkehaus. Ddd: A free graphical front-end for unix debuggers. In *ACM SIGPLAN Notices*, volume 31 of *1*, pages 22–27, 1996.